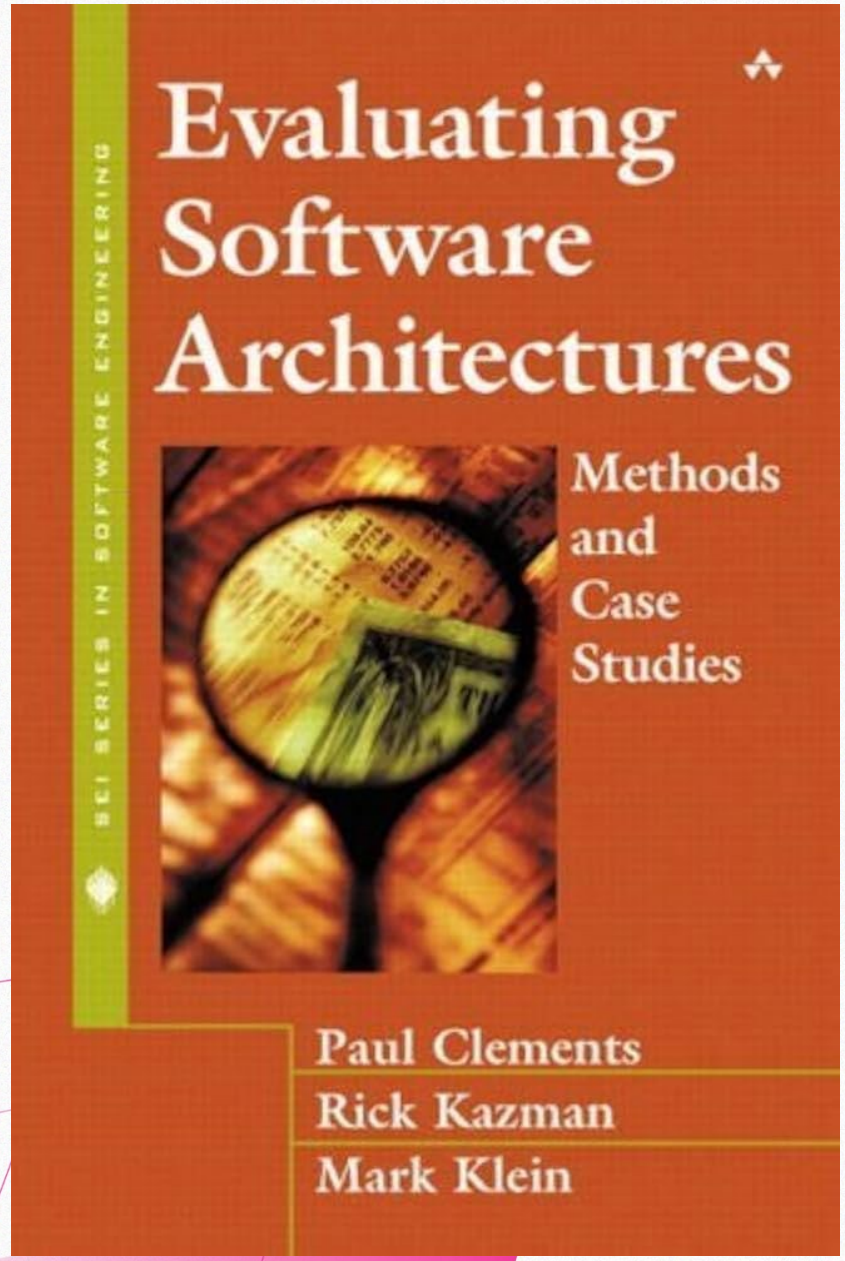
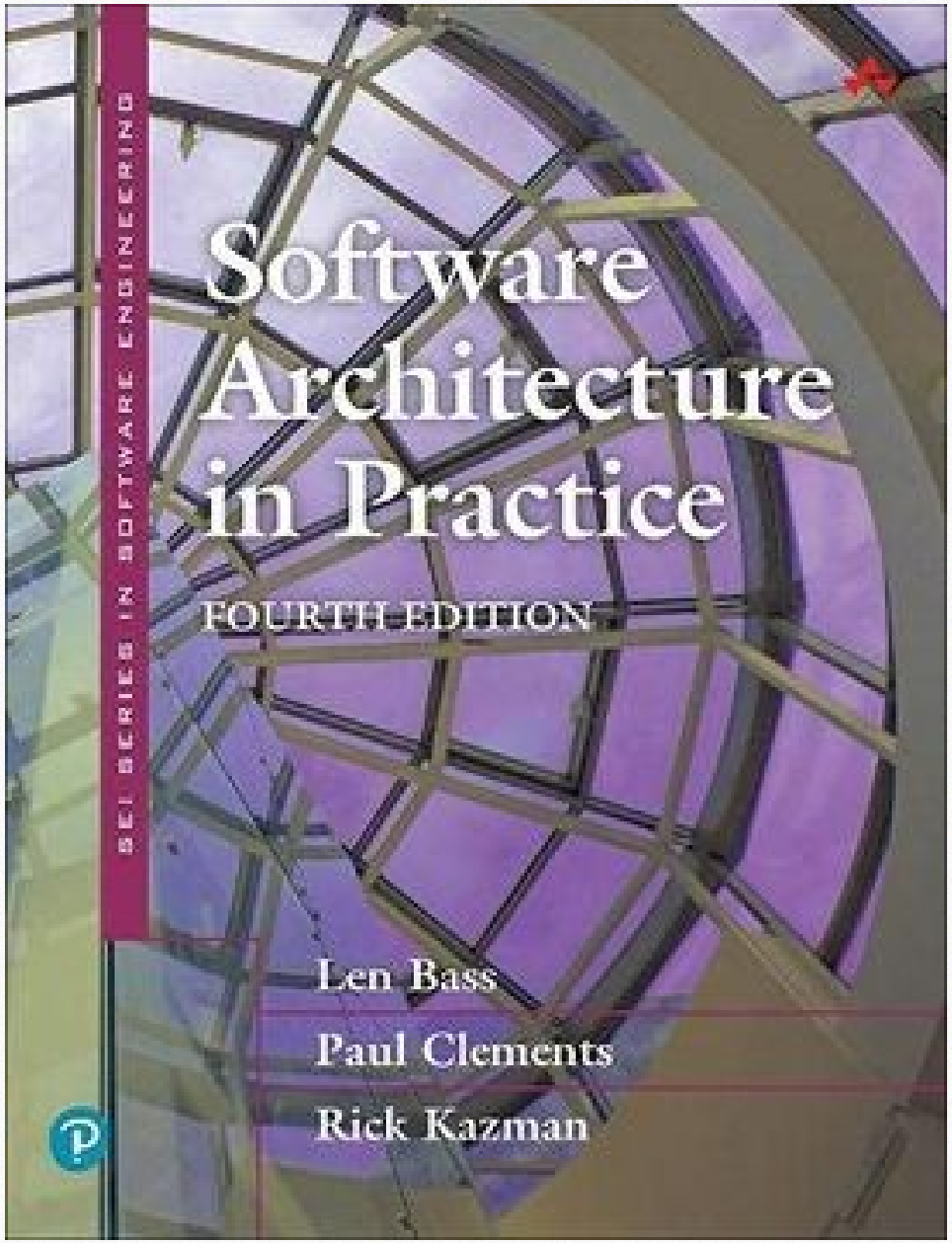
Evaluacion de la Arquitectura

“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch


SEI SERIES IN SOFTWARE ENGINEERING
Evaluating Software Architectures
Methods and Case Studies
Paul Clements
Rick Kazman
Mark Klein


SEI SERIES IN SOFTWARE ENGINEERING
Software Architecture in Practice
FOURTH EDITION
Len Bass
Paul Clements
Rick Kazman

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

ATAM:SM Method for Architecture Evaluation

CMU/SEI-2000-TR-004
ESC-TR-2000-004

Rick Kazman
Mark Klein
Paul Clements

August 2000

Product Line Systems

Unlimited distribution subject to the copyright.

A Survey on Software Architecture Analysis Methods

Liliana Dobrica and Eila Niemelä, *Member, IEEE Computer Society*

Abstract—The purpose of the architecture evaluation of a software system is to analyze the architecture to identify potential risks and to verify that the quality requirements have been addressed in the design. This survey shows the state of the research at this moment, in this domain, by presenting and discussing eight of the most representative architecture analysis methods. The selection of the studied methods tries to cover as many particular views of objective reflections as possible to be derived from the general goal. The role of the discussion is to offer guidelines related to the use of the most suitable method for an architecture assessment process. We will concentrate on discovering similarities and differences between these eight available methods by making classifications, comparisons and appropriateness studies.

Index Terms—Software architecture, analysis techniques and methods, quality attributes, scenarios.

1 Introduction

One of the major issues in software systems development today is quality. The idea of predicting the quality of a software product from a higher-level design description is not a new one. In 1972, Parnas [44] described the use of modularization and information hiding as a means of high-level system decomposition to improve flexibility and comprehensibility. In 1974, Stevens et al. [52] introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition. During recent years, the notion of software architecture (SA) has emerged as the appropriate level for dealing with software quality. This is because the scientific and industrial communities have recognized that SA sets the boundaries for the software qualities of the resulting system [7].

Recent efforts towards the systematization of the implications of using design patterns and architectural styles contribute, frequently in an informal way, to the guarantee of the quality of a design [16], [19]. It is recognized that it is not possible to measure the quality attributes of the final system based on SA design [12]. This would imply that the detailed design and implementation represent a strict projection of the architecture. The aim of analyzing the architecture of a software system is to predict the quality of a system before it has been built and not to establish precise estimates but the principal effects of an architecture [27]. The purpose of the evaluation is to analyze the SA to identify potential risks and verify that the quality requirements have been addressed in the design [38].

More formal efforts are concentrated on ensuring that the quality is addressed at the architectural level. Different communities of the software metrics, scenario-based, and attribute model-based analysts have developed their own techniques. The software metrics community has used module coupling and cohesion notions to define predictive measures of software quality [15]. Other methods include a more abstract evaluation of how the SA fulfills the domain functionality and other nonfunctional qualities [27]. Instead of presenting metrics for predictive evaluation, they exemplify the argument for performing a more qualitative or quantitative evaluation. Methods based on scenarios could be considered mature enough since they have been applied and validated over the past several years, but the development of attribute model-based architecture evaluation methods is still ongoing.

Future work is needed to develop systematic ways of bridging quality requirements of software systems with their architecture. The open problem is how to take better advantage of software architectural concepts to analyze software systems for quality attributes in a systematic and repetitive way. Being a new research domain, most of the structural methods for assessing the quality of SAs have been presented in conference and journal papers. Although refinement and experiments for validating some of the methods are ongoing, they deserve our attention because they contribute to the development of what is still an immature research area. Therefore, we decided to study such methods in order to cover as many particular points of view of objective reflections as possible to be derived from the general goal. SA is considered the first product in an architecture-based development process and, from this point of view, the analysis at this level should reveal requirement conflicts and incomplete design descriptions from a particular stakeholder's perspective. The analysis could be associated with the design of an iterative improvement of the architecture of a green software system or with the reengineering of an existent one. Prediction methods of a single quality attribute are meant to minimize

- *L. Dobrica is with the Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Spl. Independentei 313, Sect. 6, Bucharest, 77206 Romania. E-mail: liliana@cid.pub.ro.*
- *E. Niemelä is with the Software Architectures Group, Embedded Software, VTT Electronics, P.O. Box 1100 FIN-90571 Oulu, Finland. E-mail: eila.niemela@vtt.fi.*

Manuscript received 3 July 2000; revised 22 Mar. 2001; accepted 9 July 2001. Recommended for acceptance by D. Rosenblum.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112394.

0098-5589/02/$17.00 © 2002 IEEE

¿Qué Hace que una Arquitectura sea "Buena"?

¿Qué Hace que una Arquitectura sea "Buena"?

- No existe una arquitectura inherentemente buena o mala.
- La calidad de una arquitectura depende de su adecuación a un propósito específico.
- Debe alinearse con los objetivos del sistema y sus restricciones.
- Las arquitecturas pueden ser evaluadas con base en criterios bien definidos.
- Existen principios clave que ayudan a mejorar la calidad de la arquitectura.

Recomendaciones de Proceso

1. **Liderazgo y cohesión:** La arquitectura debe ser diseñada por un arquitecto o un pequeño equipo con un líder técnico claro.
2. **Enfoque en atributos de calidad:** La arquitectura debe priorizar requisitos como escalabilidad, seguridad y mantenibilidad.
3. **Documentación adecuada:** Usar vistas arquitectónicas que aborden las preocupaciones clave de los stakeholders.
4. **Evaluación continua:** Revisar la arquitectura tempranamente y de manera iterativa para garantizar su validez.
5. **Implementación incremental:** Construir un sistema esquelético funcional que permita el crecimiento progresivo.

Recomendaciones Estructurales

1. **Modularidad y encapsulación:** Definir módulos con responsabilidades claras y principios de separación de preocupaciones.
2. **Uso de patrones arquitectónicos:** Aplicar patrones bien establecidos para lograr atributos de calidad específicos.
3. **Independencia de herramientas:** Evitar dependencias fuertes con versiones específicas de productos comerciales.
4. **Separación de datos:** Mantener independientes los módulos que producen y consumen datos para mejorar la modifiabilidad.
5. **Interacciones consistentes:** Definir un número limitado de formas de interacción entre componentes para facilitar el mantenimiento y la escalabilidad.
6. **Gestón de recursos:** Identificar y controlar áreas críticas de contención de recursos, como redes o tiempo de procesamiento.

Atributos de Calidad

Principales Atributos de Calidad

Los atributos de calidad representan aspectos no funcionales del software y son claves para su evaluación. Algunos de los más importantes incluyen:

1. **Disponibilidad (Availability):** Qué tan robusto es el sistema frente a fallos.
2. **Interoperabilidad (Interoperability):** Capacidad del sistema para interactuar con otros sistemas.
3. **Modificabilidad (Modifiability):** Facilidad con la que el sistema puede evolucionar.
4. **Rendimiento (Performance):** Capacidad del sistema para manejar carga y responder en tiempo real.
5. **Seguridad (Security):** Protección contra accesos no autorizados o ataques.
6. **Testabilidad (Testability):** Facilidad para probar y validar el sistema.
7. **Usabilidad (Usability):** Facilidad de uso por parte de los usuarios finales .

Disponibilidad (Availability)

Es un atributo de calidad clave en la arquitectura de software, definido como la capacidad del sistema para estar operativo y accesible cuando se le necesita.

Es especialmente crítica en sistemas que requieren alta confiabilidad, como servidores web, sistemas financieros, aplicaciones médicas y sistemas de control en infraestructura crítica.

$$Disponibilidad = \frac{MTBF}{MTBF + MTTR}$$

MTBF (Mean Time Between Failures):
Tiempo promedio entre fallos.
MTTR (Mean Time To Repair):
Tiempo promedio para restaurar el sistema tras un fallo.

- Un sistema con **99.9% de disponibilidad** implica que está caído **8.76 horas al año**.
- Un sistema con **99.999% de disponibilidad (Five Nines)** solo tiene **5 minutos de inactividad al año**.

Mejorar la Disponibilidad

El diseño arquitectónico puede aplicar diversas **tácticas** para mejorar la disponibilidad del sistema. Estas tácticas se agrupan en tres estrategias principales:

Detección de Fallos: Antes de mitigar o recuperar un fallo, el sistema debe detectarlo de manera eficiente.

- **Monitoreo Activo:** Uso de herramientas para detectar caídas o latencias anormales (ej. Prometheus, Nagios).
- **Heartbeats:** Señales periódicas entre componentes para confirmar que están operativos.
- **Time-Outs y Watchdogs:** Si un componente no responde en un tiempo determinado, se asume que ha fallado.

Recuperación ante Fallos: Una vez detectado un fallo, el sistema debe recuperarse lo más rápido posible.

- **Failover Automático:** Si un servidor cae, otro asume su rol sin intervención manual.
- **Reinicio Automático:** Se reinicia el servicio o proceso defectuoso de manera automática.
- **Rollbacks y Snapshots:** Restauración a un estado previo funcional en caso de fallo crítico.

Prevención de Fallos: Para minimizar la probabilidad de fallos, se pueden aplicar:

- **Redundancia:** Múltiples instancias de componentes críticos (servidores, bases de datos, redes).
- **Balanceo de Carga:** Distribuir peticiones entre múltiples servidores para evitar sobrecarga.
- **Deploys Seguros (Canary Releases, Blue-Green Deployment):** Para evitar fallos en producción.

Diseño Arquitectónico para Disponibilidad

La disponibilidad no es solo un atributo de calidad aislado, sino que impacta la arquitectura general del sistema. Algunas estrategias arquitectónicas clave incluyen:

Arquitectura de Microservicios: Los microservicios permiten alta disponibilidad al distribuir cargas y evitar que un fallo afecte todo el sistema. Se suelen usar patrones como:

- **Circuit Breaker:** Evita que fallos en un servicio propaguen problemas al resto del sistema.
- **Service Mesh:** Herramientas como Istio o Linkerd permiten gestionar disponibilidad con monitoreo automático.

Arquitectura Distribuida: Un sistema distribuido puede mejorar la disponibilidad al evitar puntos únicos de fallo.

- **Bases de datos replicadas:** PostgreSQL con replicación en caliente.
- **Cluster de servidores:** Kubernetes para orquestar múltiples instancias.

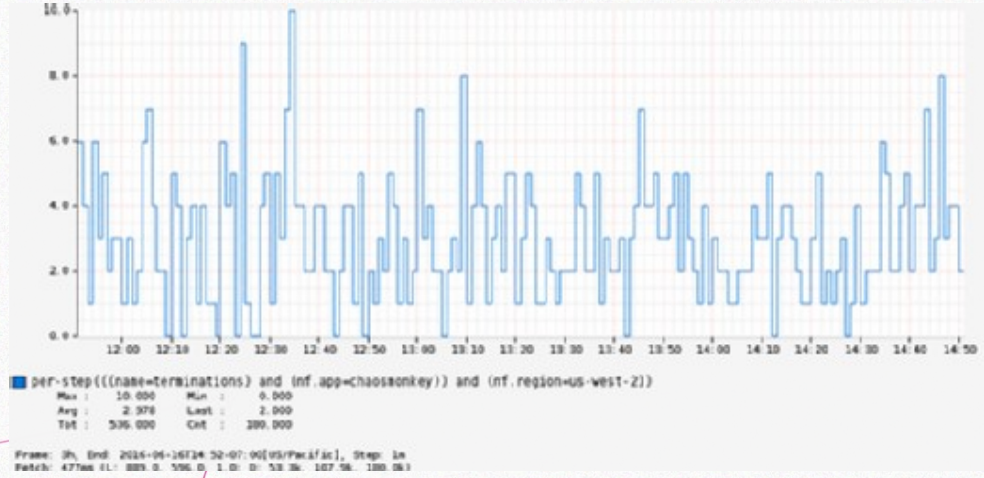
Arquitectura en la Nube: Las arquitecturas en la nube ofrecen mecanismos avanzados de disponibilidad, como:

- **Autoescalado:** AWS Auto Scaling o Google Kubernetes Engine ajustan dinámicamente recursos.
- **Regiones y zonas de disponibilidad:** AWS, Azure y GCP permiten replicar servicios en múltiples regiones geográficas.

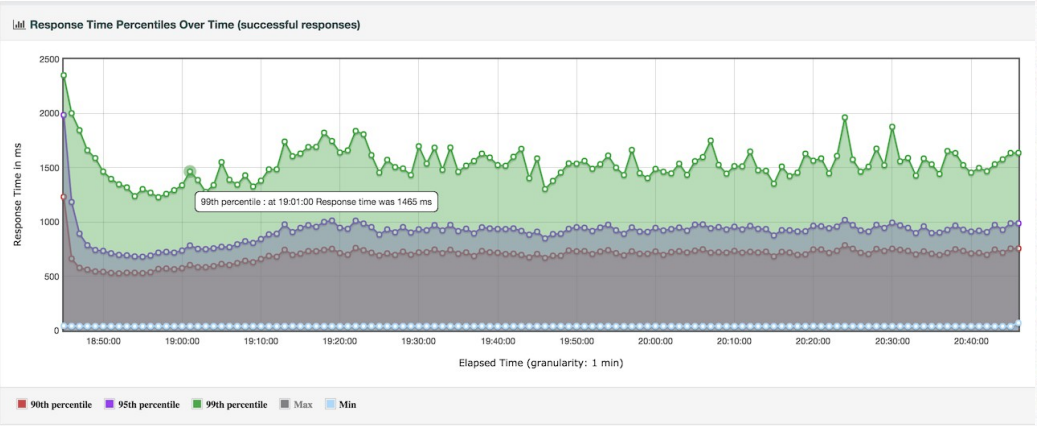
Evaluación de la Disponibilidad

Para asegurar que la arquitectura realmente cumple con los requisitos de disponibilidad, se pueden aplicar métodos como:

Pruebas de Chaos Engineering: Herramientas como Chaos Monkey simulan fallos en producción para probar la resiliencia del sistema.



Pruebas de carga y stress: Simular tráfico extremo con herramientas como JMeter.



Análisis con ATAM: Evaluar la arquitectura en base a escenarios de fallos y redundancia

Interoperabilidad (Interoperability)

La **interoperabilidad** es un atributo de calidad esencial en la arquitectura de software que define la capacidad de un sistema para interactuar eficazmente con otros sistemas, aplicaciones o dispositivos.

Es fundamental en entornos empresariales, donde múltiples sistemas deben comunicarse de manera eficiente, y en arquitecturas modernas como **microservicios, IoT y sistemas distribuidos**

Niveles de Interoperabilidad

1. **Interoperabilidad Syntáctica:** Se refiere a la compatibilidad en el formato de datos y protocolos de comunicación.

   Ejemplo: JSON, XML, REST APIs.

2. **Interoperabilidad Semántica:** Asegura que los datos compartidos sean interpretados correctamente por todas las partes.

   Ejemplo: Estándares como HL7 en salud o SWIFT en banca.

3. **Interoperabilidad Técnica:** Involucra la compatibilidad entre plataformas y sistemas operativos.

   Ejemplo: Contenedores Docker, Virtual Machines.

4. **Interoperabilidad Organizacional:** Implica políticas y procesos que permiten la integración de sistemas entre diferentes empresas o entidades.

Mejorar la Interoperabilidad

Para lograr una arquitectura interoperable, es fundamental aplicar tácticas de diseño que faciliten la integración y el intercambio de información entre sistemas. Estas tácticas pueden dividirse en cuatro categorías principales:

Uso de Estándares Comunes

El uso de estándares ampliamente adoptados facilita la comunicación entre sistemas.

Ejemplos:

- **Protocolos de comunicación:** HTTP, MQTT, WebSockets.
- **Formatos de datos:** JSON, XML, Protobuf.
- **APIs estandarizadas:** OpenAPI, GraphQL.

Middleware y Mensajería

Los sistemas pueden utilizar **middleware** o **buses de mensajería** para facilitar la comunicación entre componentes heterogéneos.

Ejemplos:

- **Middleware orientado a mensajes (MOM):** Apache Kafka, RabbitMQ.
- **Enterprise Service Bus (ESB):** MuleSoft, WSO2.
- **Service Mesh:** Istio, Linkerd.

Diseño Basado en APIs

Las APIs permiten desacoplar sistemas y facilitar la interoperabilidad. Existen diferentes enfoques:

• **APIs RESTful:** Basadas en HTTP y ampliamente utilizadas en la web.
• **APIs GraphQL:** Permiten consultas flexibles y optimizan la comunicación.
• **APIs gRPC:** Uso de comunicación binaria para mejorar rendimiento en sistemas distribuidos.

Evaluación de la Interoperabilidad

Para asegurar que un sistema es interoperable, se pueden aplicar diversas técnicas de evaluación:

Pruebas de Compatibilidad

- Validar el uso de formatos estándar como JSON, XML.
- Asegurar que las APIs cumplen con especificaciones como OpenAPI.

Análisis de Desempeño

• Medir la latencia de comunicación entre sistemas.
• Evaluar la eficiencia del middleware utilizado.

Simulación de Integraciones

• Probar la interoperabilidad con sistemas de terceros en entornos de prueba.
• Realizar pruebas de carga en servicios críticos.

Evaluación con ATAM

El *Architecture Tradeoff Analysis Method (ATAM)* permite analizar la interoperabilidad en un contexto más amplio. Se pueden considerar escenarios como:

• “Un nuevo sistema de facturación debe integrarse con el ERP actual sin modificar el código fuente.”
• “El sistema debe ser capaz de consumir APIs de múltiples proveedores sin cambios en la lógica interna.”

Rendimiento (Performance)

El **rendimiento** es un atributo de calidad fundamental en la arquitectura de software, ya que define la capacidad de un sistema para responder a solicitudes dentro de un tiempo aceptable, incluso bajo carga.

Es crítico en sistemas que manejan grandes volúmenes de datos, aplicaciones en tiempo real y servicios en la nube.

Definición y Métricas del Rendimiento

El rendimiento de un sistema se mide a través de varias métricas clave:

Métricas Claves de Rendimiento

1. **Latencia:** Tiempo que tarda el sistema en responder a una solicitud.
2. **Throughput (Rendimiento):** Cantidad de operaciones procesadas por unidad de tiempo.
3. **Tiempo de Respuesta:** Tiempo total desde que se envía una solicitud hasta que se recibe la respuesta.
4. **Escalabilidad:** Capacidad del sistema para manejar un aumento de carga sin degradación significativa.
5. **Consumo de Recursos:** Uso de CPU, memoria, ancho de banda, y disco.

Mejorar el Rendimiento

El rendimiento depende en gran medida del diseño arquitectónico. Algunas tácticas clave incluyen:

Optimización del Uso de Recursos: Reducir la carga en CPU, memoria y disco permite mejorar la velocidad del sistema.

- **Caché:** Uso de **Redis, Memcached** para almacenar resultados de consultas frecuentes.
- **Compresión de Datos:** Reducción del tamaño de los datos transmitidos (Gzip, Brotli).
- **Lazy Loading:** Carga diferida de elementos no esenciales para mejorar tiempos de respuesta.

Balanceo de Carga: Distribuir la carga entre múltiples servidores evita cuellos de botella.

- **Load Balancers:** Nginx, HAProxy, AWS ELB distribuyen tráfico eficientemente.
- **Round Robin vs. Least Connections:** Diferentes estrategias para asignar solicitudes.

Concurrencia y Paralelismo: Permitir múltiples procesos o hilos de ejecución mejora el throughput.

- **Programación Asíncrona:** Uso de **async/await, Promises** en Node.js o Python.
- **Multiprocesamiento:** Ejecutar tareas en paralelo con **procesos en lugar de hilos**.

Optimización de Base de Datos: El acceso a la base de datos suele ser el mayor cuello de botella.

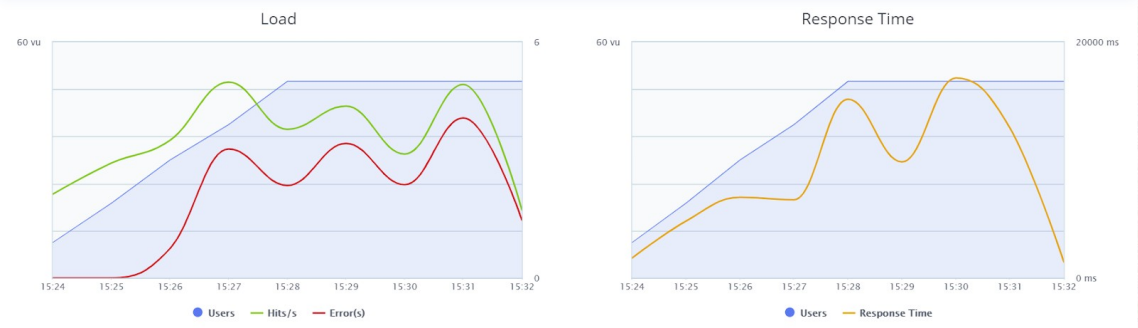
- **Índices:** Mejoran la velocidad de búsqueda en bases de datos relacionales.
- **Denormalización:** Reducción de joins en bases de datos NoSQL para mejorar lectura.
- **Replicación y Sharding:** Distribución de datos entre múltiples nodos.

Evaluación del Rendimiento

Para garantizar que el sistema cumple con los requisitos de rendimiento, se deben realizar pruebas y análisis:

Pruebas de Carga: Simulan tráfico para evaluar la capacidad del sistema (JMeter, Gatling).

Pruebas de Estrés: Miden el comportamiento del sistema bajo condiciones extremas.



Pruebas de Capacidad: Evalúan el número máximo de usuarios soportados sin degradación.

Seguridad (Security)

La **seguridad** es un atributo de calidad crítico en la arquitectura de software, ya que garantiza la protección del sistema contra accesos no autorizados, alteraciones maliciosas y filtraciones de datos.

Un diseño arquitectónico seguro debe prever amenazas desde el principio y aplicar medidas que minimicen vulnerabilidades.

Principios de Seguridad en Arquitectura

El diseño seguro de software sigue principios fundamentales para reducir riesgos:

Principio del Mínimo Privilegio: Cada componente o usuario solo debe tener los permisos estrictamente necesarios para su función.

Ejemplo:

- **Aplicaciones web:** Evitar que el servidor de la aplicación tenga acceso directo a la base de datos sin restricciones.
- **Sistemas operativos:** Aplicar control de acceso basado en roles (RBAC).

Principio de Defensa en Profundidad: No depender de un solo mecanismo de seguridad, sino usar múltiples capas de protección.

Ejemplo:

- **Firewalls + Autenticación + Cifrado**: Aunque un atacante eluda una capa, otras siguen protegiendo el sistema.

Seguridad por Diseño: El sistema debe ser seguro desde la fase de diseño, evitando depender de parches o medidas reactivas. Técnicas:

- **Threat Modeling (Modelado de Amenazas):** Identificar vectores de ataque antes del desarrollo.
- **Zero Trust Architecture:** No confiar en ninguna entidad dentro o fuera de la red sin autenticación y autorización.

Seguridad por Defecto: El software debe ser seguro en su configuración inicial, sin requerir ajustes manuales.

Ejemplo:

- **Bases de datos:** Deshabilitar accesos remotos no autenticados por defecto.

Mejorar la Seguridad

La arquitectura de software puede incluir diversas tácticas de seguridad para mitigar riesgos.

Control de Acceso y Autenticación

- **Autenticación Multifactor (MFA):** Combinar contraseñas con OTPs, biometría, etc.
- **OAuth 2.0 / OpenID Connect:** Estandarización de autenticación basada en tokens.
- **Gestión de Sesiones Segura:** Evitar ataques de secuestro de sesiones con expiración y revocación de tokens.

Protección de Datos

- **Cifrado en Tránsito:** Uso de TLS/SSL para proteger datos en redes.
- **Cifrado en Reposo:** Bases de datos cifradas (AES-256).
- **Tokenización:** Reemplazo de datos sensibles con tokens irreversibles.

Detección y Respuesta a Intrusos

- **Monitoreo de Logs:** Análisis en tiempo real con herramientas como SIEM (Security Information and Event Management).
- **IDS/IPS (Intrusion Detection/Prevention Systems):** Firewalls avanzados que detectan y bloquean ataques.

Seguridad en el Desarrollo

- **Validación de Entradas:** Evitar inyección SQL y XSS con sanitización de datos.
- **Code Reviews y Pruebas de Seguridad:** Uso de herramientas como SAST y DAST para identificar vulnerabilidades.

Evaluación de la Seguridad

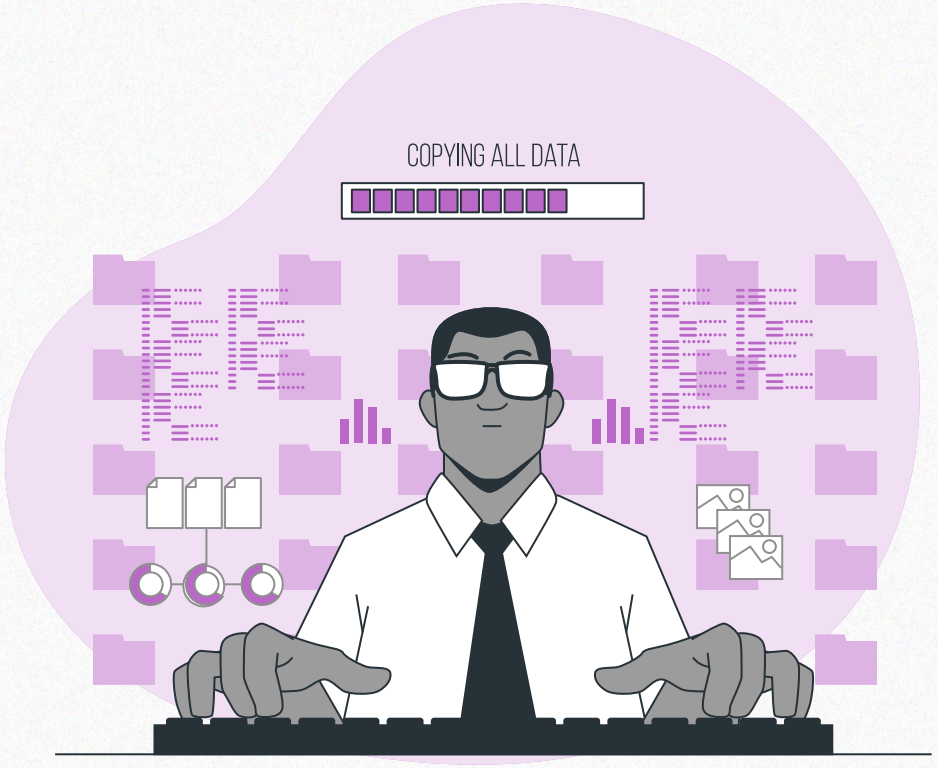
Para garantizar que la arquitectura es segura, se deben realizar evaluaciones y pruebas de seguridad.

Pruebas de Seguridad

- **Pentesting:** Simulación de ataques para detectar vulnerabilidades.
- **Análisis Estático (SAST):** Identificación de fallos en código fuente antes de su ejecución.
- **Análisis Dinámico (DAST):** Pruebas en aplicaciones en ejecución.

Auditoría y Cumplimiento

- **ISO 27001, NIST, PCI-DSS:** Estándares de seguridad para garantizar conformidad.
- **Revisión de Logs:** Análisis de actividad sospechosa para prevenir ataques.

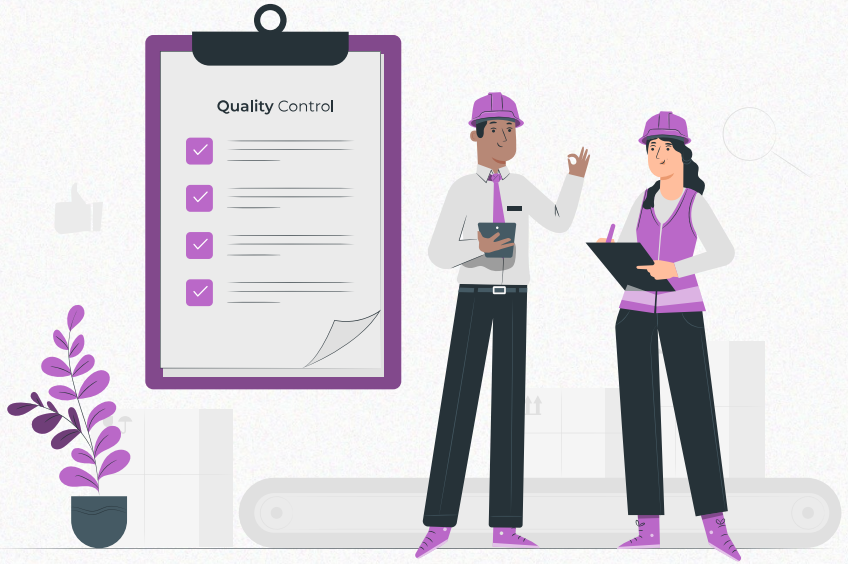


Testabilidad (Testability)

La **testabilidad** es un atributo de calidad fundamental en la arquitectura de software que mide qué tan fácil es probar un sistema.

Una arquitectura bien diseñada debe permitir la detección rápida de defectos, la automatización de pruebas y la identificación eficiente de la causa raíz de errores.

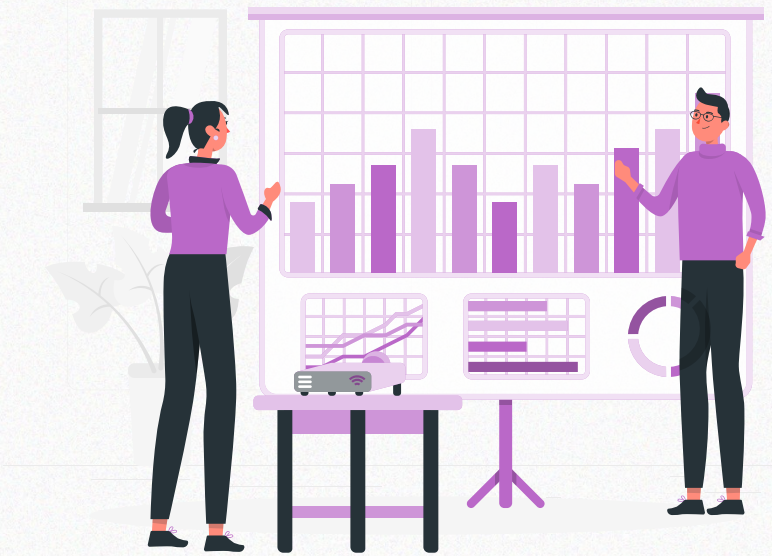
Una arquitectura **altamente testeable** reduce el costo de mantenimiento y mejora la calidad del software al facilitar la implementación de pruebas automatizadas y la detección temprana de defectos



Métricas de Testabilidad

La testabilidad mide qué tan fácil es verificar que el sistema funciona correctamente. Se evalúa mediante varias métricas clave:

1. **Observabilidad:** Facilidad para inspeccionar el estado interno del sistema durante las pruebas.
2. **Controlabilidad:** Capacidad de forzar el sistema a estados específicos para probar escenarios particulares.
3. **Automatización de Pruebas:** Capacidad del sistema para soportar pruebas automatizadas sin intervención manual.
4. **Aislamiento de Componentes:** Facilidad para probar partes individuales del sistema sin dependencias externas.
5. **Tiempo de Ejecución de Pruebas:** Tiempo requerido para ejecutar la suite de pruebas completa.



Mejorar la Testabilidad

El diseño arquitectónico influye directamente en la facilidad de prueba. Algunas tácticas clave incluyen:

Desacoplamiento de Componentes: Reducir dependencias entre módulos facilita la prueba aislada de cada componente.

- **Principio de Inversión de Dependencias (DI):** Inyectar dependencias en lugar de acoplarlas directamente.
- **Uso de Interfaces:** Permitir sustitución de implementaciones en pruebas.

Diseño Modular: Dividir el sistema en módulos pequeños y bien definidos facilita la prueba de cada unidad de manera independiente.

- **Microservicios:** Permiten probar cada servicio de forma aislada.
- **Patrón MVC:** Facilita la prueba separada de lógica de negocio, presentación y datos.

Exposición de Puntos de Observabilidad: Permitir inspeccionar el estado del sistema facilita la detección de fallos.

- **Logging estructurado:** Uso de herramientas como ELK Stack o Splunk.
- **Tracing distribuido:** OpenTelemetry para rastrear llamadas entre servicios.

Evaluación de la Testabilidad

Para medir qué tan testeable es un sistema, se pueden aplicar diversas estrategias de evaluación.

Cobertura de Código: Mide qué porcentaje del código fuente es ejecutado durante las pruebas.

• **Herramientas:** JaCoCo (Java), Istanbul (JavaScript), Coverage.py (Python).

JaCoCo > org.jacoco.report

org.jacoco.report

Element	Instruction Coverage		Missed Classes	Missed Methods	Missed Blocks	Missed Lines
org.jacoco.report.html		63%	10 / 20	54 / 128	84 / 214	117 / 385
org.jacoco.report.csv		20%	8 / 9	35 / 43	52 / 68	100 / 126
org.jacoco.report		75%	3 / 7	6 / 25	12 / 69	26 / 98
org.jacoco.report.xml		90%	2 / 10	9 / 42	13 / 88	17 / 146
org.jacoco.report.html.resources		87%	1 / 3	1 / 7	9 / 40	2 / 35
Total		64%	24 / 49	105 / 245	170 / 479	262 / 790

Code Coverage Report for JaCoCo 0.1.0.20091027174426 — Created with JaCoCo 0.1.0.20091027174426

Pruebas de Integración Automatizadas: Validar la interacción entre múltiples componentes mediante pruebas automatizadas.

• **Ejemplo:** Pruebas de API con Postman, REST Assured.

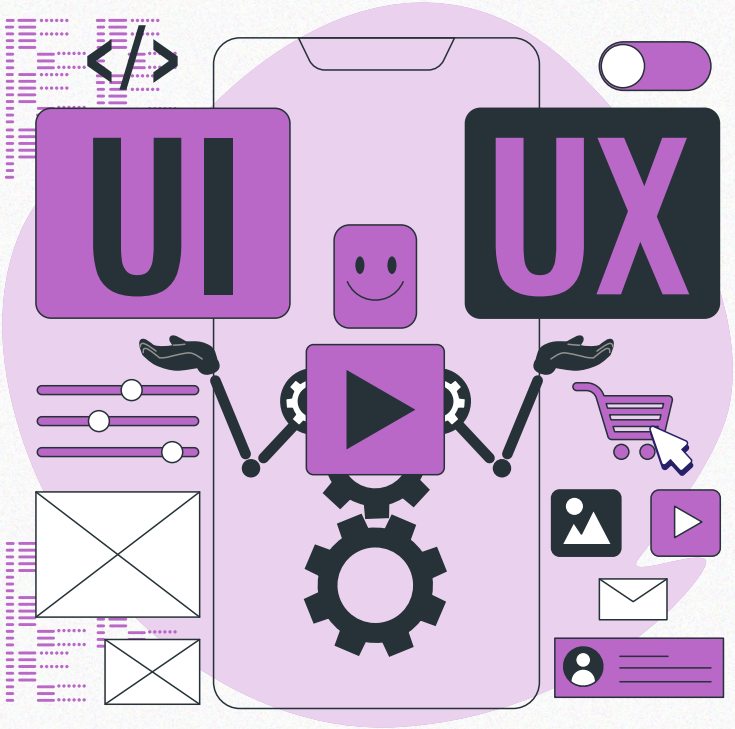
```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
            get("/lotto/{id}", 5).
    then().
            statusCode(200).
            body("lotto.lottoId", equalTo(5),
                 "lotto.winners.winnerId", hasItems(23, 54));

}
```

Usabilidad (Usability)

Es un atributo de calidad que mide qué tan fácil es para los usuarios interactuar con un sistema.

Se centra en la experiencia del usuario (UX), asegurando que la interfaz sea intuitiva, accesible y eficiente.

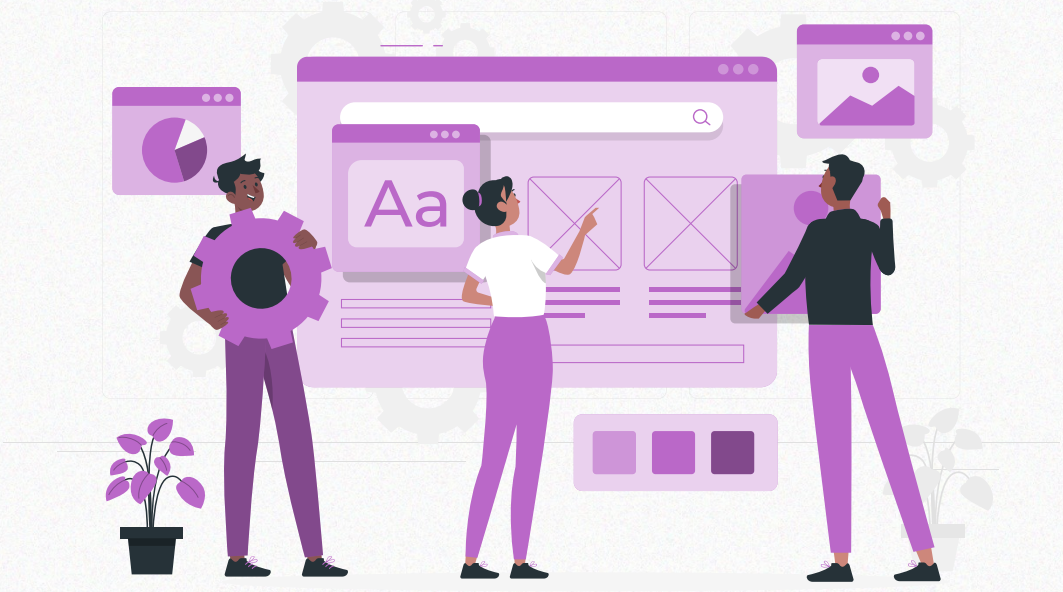
Una arquitectura bien diseñada debe considerar la usabilidad desde el principio para evitar sistemas complejos o difíciles de aprender.



Métricas de Usabilidad

La usabilidad se evalúa a través de varios factores clave:

1. **Facilidad de Aprendizaje:** Qué tan rápido un usuario nuevo puede entender y operar el sistema.
2. **Eficiencia de Uso:** Qué tan rápido los usuarios pueden completar tareas.
3. **Satisfacción del Usuario:** Qué tan cómodo y agradable es el uso del sistema.
4. **Prevención de Errores:** Cómo la arquitectura reduce la posibilidad de que los usuarios cometan errores.
5. **Recuperabilidad de Errores:** Qué tan fácil es para un usuario corregir un error sin soporte externo.



Mejorar la Usabilidad

La arquitectura de software puede influir en la usabilidad al proporcionar estructuras que faciliten la interacción y reduzcan la complejidad.

Consistencia en la Interfaz

- **Uso de patrones de diseño estándar:** Interfaces que sigan principios comunes como Material Design o Human Interface Guidelines.
- **Estructura homogénea:** Los mismos elementos deben comportarse igual en todo el sistema.

Retroalimentación Instantánea

- **Mensajes de error claros:** Indicar qué salió mal y cómo corregirlo.
- **Indicadores de carga:** Reducen la incertidumbre cuando un proceso tarda en completarse.

Minimización de Pasos: Reducir la cantidad de interacciones necesarias para completar una tarea:

- **Autocompletado y predicciones inteligentes.**
- **Default settings bien configurados** para evitar configuraciones manuales innecesarias.

Diseño Accesible: Asegurar que el sistema sea utilizable por la mayor cantidad de usuarios posibles:

- **Compatibilidad con lectores de pantalla** para usuarios con discapacidades visuales.
- **Altos contrastes y tamaños de fuente adecuados** para mejorar la legibilidad.

Personalización y Configurabilidad: Permitir que los usuarios adapten el sistema a sus necesidades:

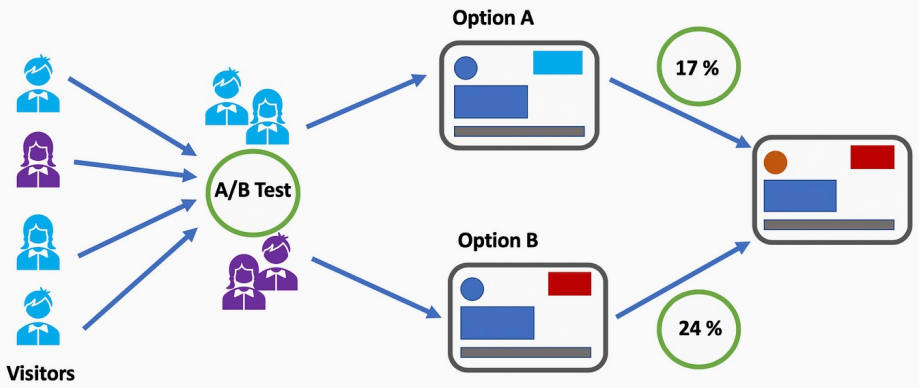
- Dark mode, ajustes de accesibilidad y layouts flexibles.
- Dashboards personalizables según el rol del usuario.

Evaluación de la Usabilidad

Para garantizar una buena experiencia de usuario, se deben realizar evaluaciones continuas.

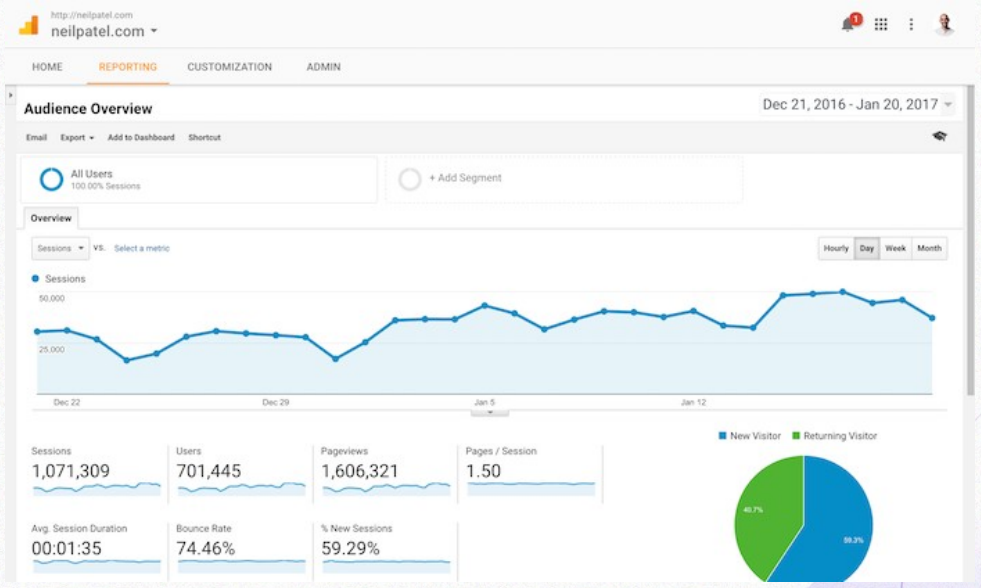
Pruebas de Usabilidad

- **Pruebas con usuarios reales:** Observar cómo interactúan con el sistema.
- **A/B Testing:** Comparar diferentes versiones de una interfaz para determinar cuál es más efectiva.



Análisis de Datos y Telemetría

- **Heatmaps:** Herramientas como Hotjar muestran qué partes de la interfaz reciben más interacción.
- **Analytics de Uso:** Google Analytics o Mixpanel permiten rastrear comportamiento del usuario.



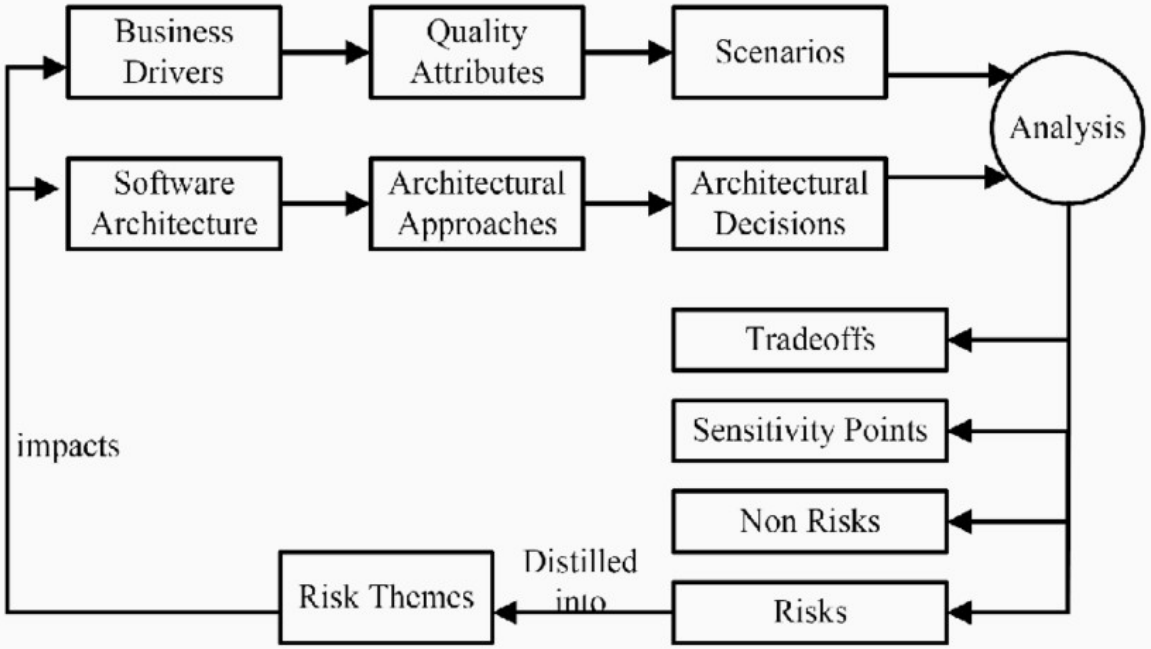
El metodo ATAM

ATAM
(Architecture Tradeoff Analysis Method)

El método **Attribute-based Architectural Tactics and Models (aTAM)** se basa en la identificación de **tácticas arquitectónicas** que influyen en los atributos de calidad del sistema.

Su propósito es proporcionar un análisis sistemático de cómo la arquitectura del software satisface los requisitos no funcionales.

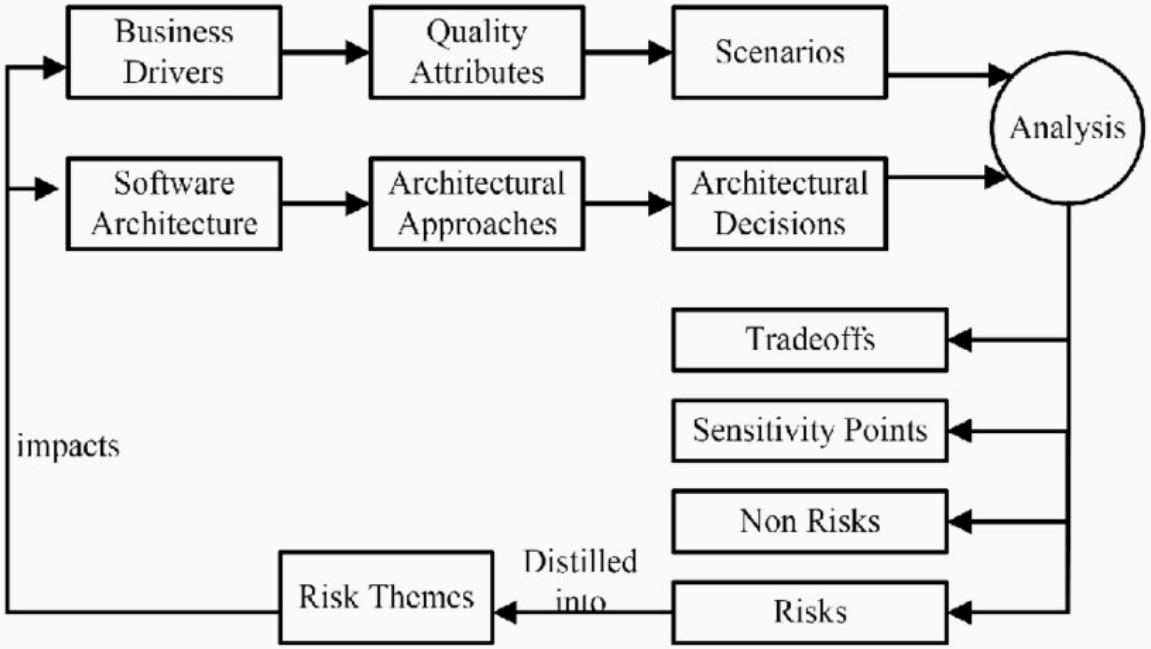
- Se priorizan los atributos según la importancia para los stakeholders (usuarios, desarrolladores, arquitectos y clientes).
- Se utilizan técnicas como encuestas y entrevistas para determinar las necesidades del negocio.



ATAM
(Architecture Tradeoff Analysis Method)

Factores de Negocio y Calidad:

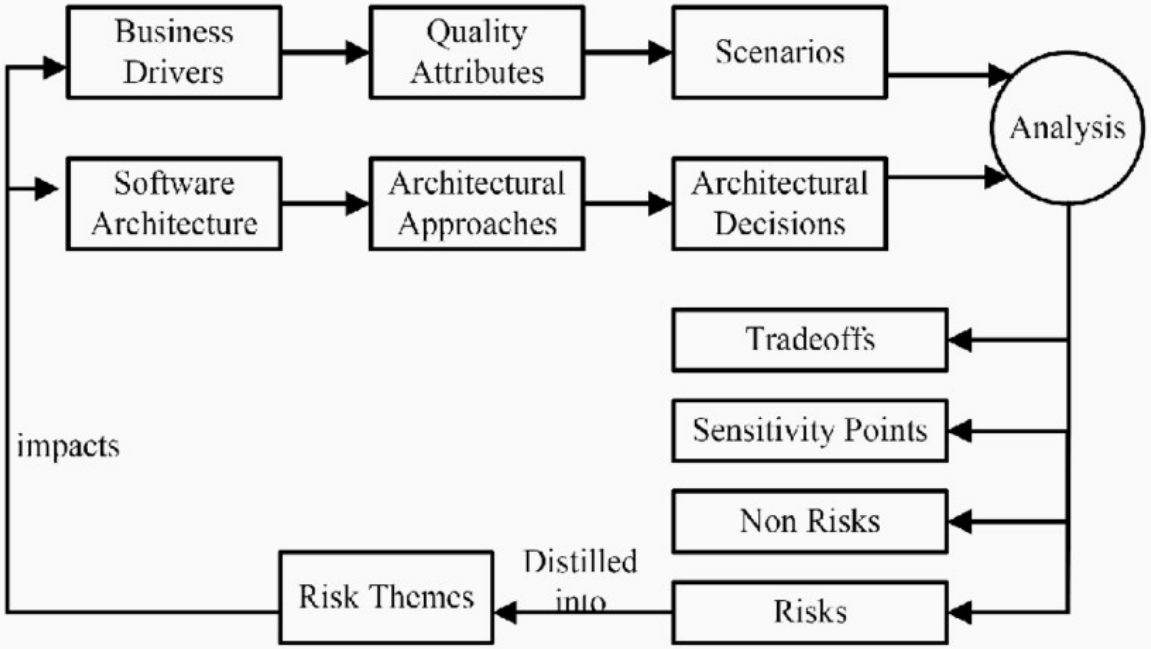
- **Business Drivers (Impulsores de Negocio):** ¿Cuáles son las metas y prioridades del negocio?
- **Quality Attributes (Atributos de Calidad):** ¿Qué características del sistema son más importantes (rendimiento, seguridad, escalabilidad, etc.)?
- **Scenarios (Escenarios de Uso) :** ¿Cómo se espera que funcione la arquitectura en diferentes situaciones?



ATAM
(Architecture Tradeoff Analysis Method)

Elementos de Arquitectura:

- **Software Architecture (Arquitectura del Software):** Modelo actual del sistema y su estructura.
- **Architectural Approaches (Enfoques Arquitectónicos) :** Estrategias utilizadas para cumplir con los objetivos de calidad.
- **Architectural Decisions (Decisiones Arquitectónicas) :** Elecciones específicas sobre cómo estructurar la arquitectura (ej. microservicios vs. monolito).



Fases Clave: Identificación de Atributos de Calidad

El método **Attribute-based Architectural Tactics and Models (aTAM)** se basa en la identificación de **tácticas arquitectónicas** que influyen en los atributos de calidad del sistema.

Su propósito es proporcionar un análisis sistemático de cómo la arquitectura del software satisface los requisitos no funcionales.

- **Rendimiento**
- **Escalabilidad**
- **Disponibilidad**
- **Seguridad**
- **Mantenibilidad**
- **Interoperabilidad**

Fases Clave: Análisis de Tácticas Arquitectónicas

Se identifican las **tácticas arquitectónicas** utilizadas en el sistema para mejorar los atributos de calidad. Algunas tácticas comunes incluyen:

Atributo de Calidad	Táctica Arquitectónica
Rendimiento	Caching, Load Balancing, Parallel Processing
Escalabilidad	Microservicios, Elasticidad en la nube
Disponibilidad	Replicación, Failover, Heartbeat Monitoring
Seguridad	Cifrado, Autenticación, Control de Acceso
Mantenibilidad	Modularidad, Encapsulación, Uso de Frameworks
Interoperabilidad	API REST, Estándares de comunicación (SOAP, GraphQL)

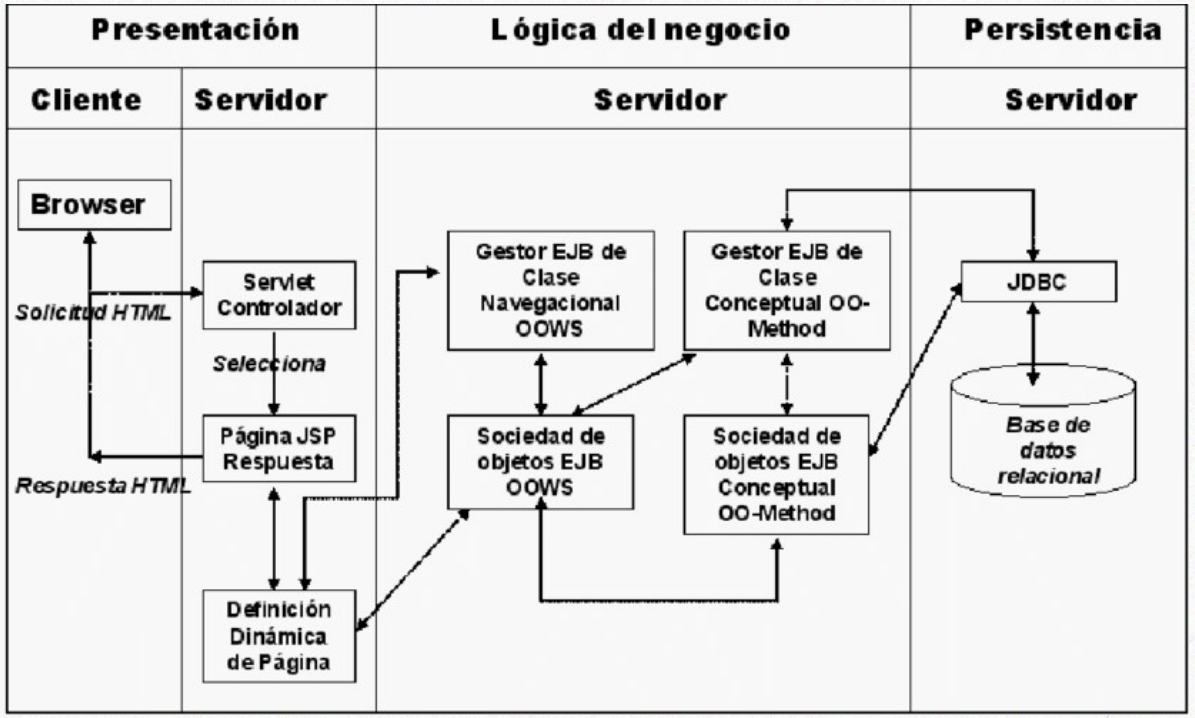
Fases Clave: Modelado de la Arquitectura

Se generan representaciones de la arquitectura en diferentes vistas para facilitar el análisis:

- **Vista de módulos**: Organización de los componentes del software.
- **Vista de concurrencia**: Gestión de procesos y hilos.
- **Vista de despliegue**: Infraestructura física y lógica del sistema.
- **Vista de datos**: Flujo de información y persistencia.

Se analizan los patrones arquitectónicos utilizados, como:

- Monolito
- Microservicios
- Event-driven architecture
- Arquitectura en capas



Fases Clave: Evaluación de Escenarios de Calidad

• Se diseñan **escenarios de evaluación** que prueban cómo la arquitectura responde a diferentes condiciones.
• Se pueden usar simulaciones, pruebas de carga o experimentos controlados.

Ejemplo de escenarios:

• **Rendimiento**: ¿Cómo se comporta el sistema con 10,000 usuarios simultáneos?
• **Escalabilidad**: ¿El sistema puede escalar horizontalmente sin degradar la experiencia del usuario?
• **Disponibilidad**: ¿Qué ocurre si falla un nodo en la arquitectura de microservicios?
• **Seguridad**: ¿Cómo responde el sistema ante un ataque de inyección SQL?

Escenario de Calidad	Pregunta Clave	Evaluación
Escalabilidad	¿La plataforma soporta 10,000 usuarios simultáneos?	Simulación de carga con JMeter
Disponibilidad	¿Qué pasa si falla un servidor?	Prueba de failover con AWS Auto Scaling
Rendimiento	¿Cuánto tarda en cargar un curso con videos?	Pruebas de latencia con CDN activado y sin activar
Seguridad	¿Es posible acceder a datos de otro usuario?	Pentesting con OWASP ZAP

Ejemplo de Resultados:

- **Escalabilidad:** Sin optimización, **Amazon RDS** comienza a ralentizarse con 5,000 usuarios. Se necesita particionado de base de datos.
- **Disponibilidad:** AWS Auto Scaling funcionó correctamente.
- **Rendimiento:** Carga de videos en <2s con **CDN**, pero sin él sube a 8s.
- **Seguridad:** Prueba de pentesting detectó un endpoint sin validación correcta.

Fases Clave: Identificación de Riesgos y Recomendaciones

- Se identifican posibles **riesgos arquitectónicos**, como:
    - Cuellos de botella en la base de datos.
    - Falta de redundancia en servidores críticos.
    - Ausencia de autenticación segura en las APIs.
- Se priorizan los riesgos en función de su impacto y probabilidad.
- Se generan **recomendaciones** para mitigar los riesgos y mejorar la arquitectura.

Ejemplo de recomendaciones:

- Implementar caching en Redis para mejorar la velocidad de respuesta.
- Usar balanceadores de carga para distribuir la carga entre servidores.
- Aplicar OAuth 2.0 para mejorar la seguridad de las APIs.

El metodo CBAM

CBAM (Cost-Benefit Analysis Method)

CBAM
(Cost-Benefit Analysis Method)

El **CBAM (Cost-Benefit Analysis Method)** es un método para evaluar **decisiones arquitectónicas** basándose en el análisis costo-beneficio.

Su objetivo es ayudar a los arquitectos de software a seleccionar alternativas que maximicen la calidad del sistema con una inversión óptima de recursos.

Este método es útil cuando hay múltiples opciones arquitectónicas y se deben tomar decisiones informadas considerando **factores técnicos y económicos**.

Fases del Método CBAM

Fase 1: Identificación de Atributos de Calidad y Decisiones Arquitectónicas

Identificar **qué atributos de calidad** son más importantes para el sistema y **qué decisiones arquitectónicas afectan estos atributos**.

Pasos:

1. Identificar los **atributos de calidad clave** para el sistema (rendimiento, escalabilidad, seguridad, disponibilidad, etc.).
2. Analizar cómo las **decisiones arquitectónicas** impactan estos atributos.
3. Involucrar a los **stakeholders** (clientes, usuarios, arquitectos, gerentes) para priorizar los atributos según su **impacto en el negocio**.

Si el sistema es una **plataforma de e-learning**, los atributos importantes pueden ser:

- **Escalabilidad** (para soportar más usuarios).
- **Disponibilidad** (para evitar caídas en clases en vivo).
- **Seguridad** (para proteger datos de usuarios).

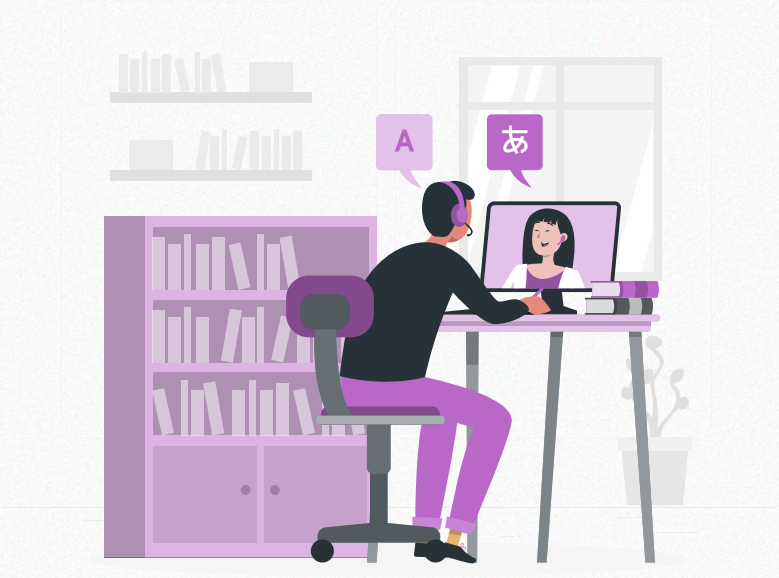
Fases del Método CBAM

Fase 2: Priorización de Escenarios de Calidad

Definir y priorizar **escenarios de calidad**, considerando su impacto en el negocio y la arquitectura.

Pasos:

1. Para cada **atributo de calidad**, definir escenarios de evaluación.
2. Asignar **puntuaciones** a cada escenario basado en dos factores:
    - **Impacto en el negocio** (1 a 5)
    - **Importancia técnica** (1 a 5)
3. Calcular una **prioridad total** sumando ambas puntuaciones.



Escenario de Calidad	Impacto en el Negocio	Importancia Técnica	Prioridad Total
Soporte para 10,000 usuarios simultáneos	5	5	**10**
Clases en vivo sin interrupciones	5	4	**9**
Protección de datos del usuario	3	5	**8**

Fases del Método CBAM

Fase 3: Identificación de Alternativas Arquitectónicas

Definir posibles **soluciones arquitectónicas** para mejorar los atributos de calidad identificados.

Pasos:

1. Proponer **múltiples soluciones** para cada escenario crítico.
2. Identificar **tácticas arquitectónicas** aplicables (caching, load balancing, microservicios, CDNs, etc.).
3. Listar las ventajas y desventajas de cada alternativa.

Escenario	Alternativa A	Alternativa B
Soporte 10,000 usuarios	Load Balancer	Kubernetes + Auto Scaling
Disponibilidad	Réplicas en base de datos	Multi-región en AWS
Seguridad	OAuth 2.0	Cifrado AES-256

Fases del Método CBAM

Fase 4: Estimación de Beneficios, Costos y Riesgos

Cuantificar el **beneficio esperado**, el **costo de implementación** y los **riesgos técnicos** de cada alternativa.

Pasos:

1. **Estimar el beneficio** de cada alternativa en una escala de 1 a 5.
2. **Calcular el costo de implementación** (puede ser en dinero o esfuerzo en horas/personas).
3. **Evaluar el riesgo técnico** (cuán difícil es implementar la solución, en una escala de 1 a 5).

Alternativa	Beneficio Esperado (1-5)	Costo Estimado ($)	Riesgo Técnico (1-5)
Load Balancer	**4**	$5,000	2
Kubernetes + Auto Scaling	**5**	$50,000	4
Microservicios	**5**	$100,000	5

Fases del Método CBAM

Fase 5: Cálculo del Retorno de Inversión (ROI)

Determinar qué opción es **más rentable** calculando su **ROI (Return on Investment).**

$$ROI = \frac{\text{Beneficio} - \text{Costo}}{\text{Costo}}$$

Alternativa	Beneficio	Costo ($)	ROI (%)
Load Balancer	4	$5,000	**80%**
Kubernetes + Auto Scaling	5	$50,000	**0%**
Microservicios	5	$100,000	**-50%**

Fases del Método CBAM

Fase 6: Toma de Decisiones y Plan de Implementación

Seleccionar la mejor alternativa y definir **cómo se implementará**.

Pasos:

1. **Elegir la opción con mejor ROI** y alineada con los objetivos del negocio.
2. **Planificar la implementación** con cronogramas y responsables.
3. **Definir métricas** para medir el impacto de la decisión tomada.

Ejemplo de Plan de Implementación:

- **Etapa 1 (1 mes)** → Implementar Load Balancer en AWS.
- **Etapa 2 (2 meses)** → Monitorear impacto y ajustar configuración.
- **Etapa 3 (6 meses)** → Revisar si se necesita una solución más robusta (como Kubernetes).

SAAM

Scenario-Based Architecture Analysis Method

SAAM (Scenario-Based Architecture Analysis Method)

El **SAAM (Scenario-Based Architecture Analysis Method)** es un método estructurado para evaluar **arquitecturas de software** basado en escenarios de uso.

Su objetivo es determinar la capacidad de la arquitectura para **soportar cambios futuros** y cumplir con los requisitos de calidad.

¿Por qué usar SAAM?

- Evaluar la **modificabilidad** y **mantenibilidad** de una arquitectura.
- Analizar el **impacto de posibles cambios** en el sistema.
- Comparar **diferentes arquitecturas** para determinar cuál es más flexible.
- Identificar **riesgos** en el diseño arquitectónico antes de la implementación.

Fases del Método SAAM

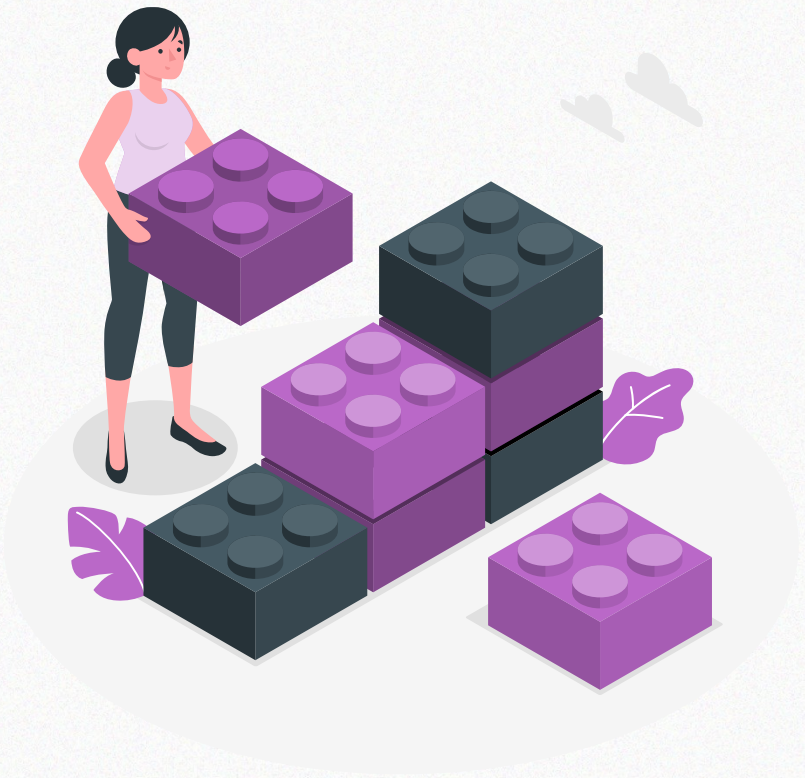
El método **SAAM** sigue cinco fases:

Fase 1: Definición del Modelo Arquitectónico

Representar la arquitectura del sistema en términos de **componentes y conexiones**.

Pasos:

1. **Identificar los componentes principales** de la arquitectura (módulos, servicios, bases de datos, interfaces, etc.).
2. **Definir las conexiones entre los componentes**, describiendo cómo interactúan entre sí.
3. **Especificar los atributos de calidad** que se desean evaluar (modificabilidad, rendimiento, seguridad, etc.).
4. **Dibujar una representación gráfica** de la arquitectura.



Fases del Método SAAM

El método **SAAM** sigue cinco fases:

Fase 2: Definición de Escenarios de Evaluación

Definir **escenarios de uso** que podrían afectar la arquitectura en el futuro.

Pasos:

1. Identificar **escenarios de cambios funcionales** (nuevas funcionalidades o modificaciones en las existentes).
2. Identificar **escenarios de cambios no funcionales** (mejoras en rendimiento, escalabilidad, seguridad, etc.).
3. Incluir **escenarios de fallos** o errores inesperados.

Escenario	Tipo
Agregar autenticación con Google y Facebook	Cambio funcional
Mejorar rendimiento en clases en vivo para 5,000 usuarios simultáneos	Cambio no funcional
Recuperar datos después de una caída del servidor	Escenario de fallo

Fases del Método SAAM

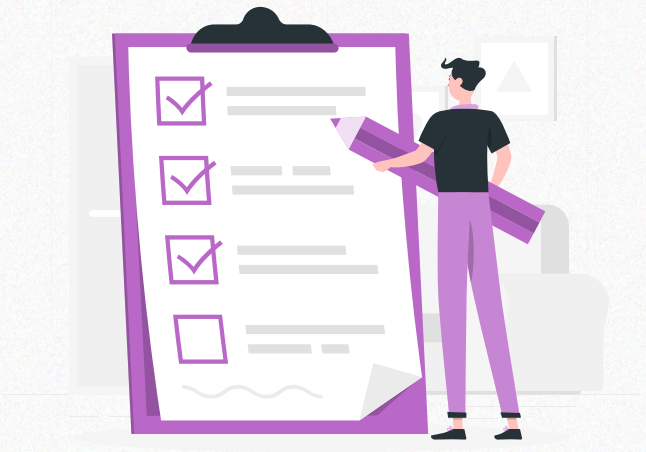
El método **SAAM** sigue cinco fases:

Fase 3: Clasificación y Priorización de Escenarios

Ordenar los escenarios en función de **su impacto y frecuencia de ocurrencia**.

Pasos:

1. **Asignar valores de impacto y frecuencia** a cada escenario.
   - **Impacto** (1-5): Qué tanto afecta el sistema.
   - **Frecuencia** (1-5): Probabilidad de que ocurra.
2. **Calcular la prioridad** sumando ambos valores.



Escenario	Impacto (1-5)	Frecuencia (1-5)	Prioridad Total
Agregar autenticación con Google y Facebook	4	5	**9**
Mejorar rendimiento en clases en vivo	5	4	**9**
Recuperar datos tras una caída	5	3	**8**

Fases del Método SAAM

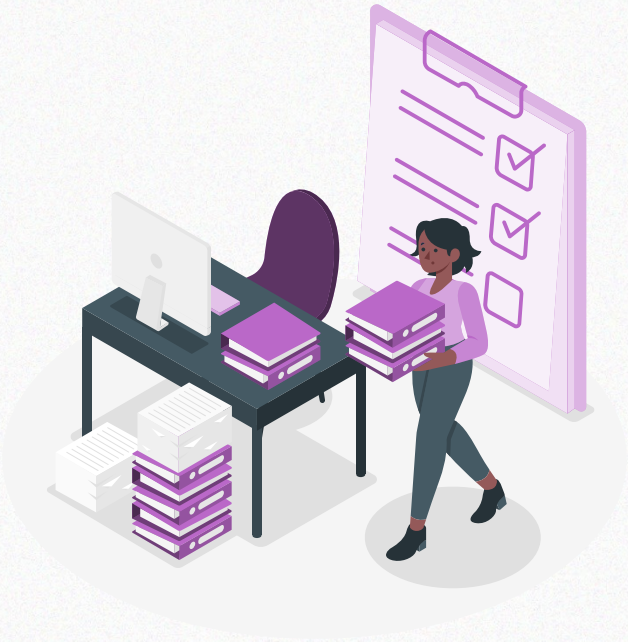
El método **SAAM** sigue cinco fases:

Fase 4: Evaluación del Impacto en la Arquitectura

Determinar **qué tan difícil es implementar cada escenario** en la arquitectura actual.

Pasos:

1. **Analizar cuántos componentes se ven afectados** por cada escenario.
2. **Evaluar la complejidad de los cambios** necesarios en cada componente.
3. **Calcular el esfuerzo de implementación** en horas-persona o costo estimado.



Escenario	Componentes Afectados	Complejidad (1-5)
Agregar autenticación con Google y Facebook	3	3
Mejorar rendimiento en clases en vivo	5	4
Recuperar datos tras una caída	2	5

Fases del Método SAAM

El método **SAAM** sigue cinco fases:

Fase 5: Análisis de Resultados y Recomendaciones

Identificar **riesgos arquitectónicos** y proponer soluciones para mejorar la arquitectura.

Pasos:

1. Identificar **componentes críticos** que dificultan la modificabilidad.
2. Proponer **tácticas arquitectónicas** para mejorar la flexibilidad.
3. Generar un **plan de acción** para optimizar la arquitectura.

Problema Detectado	Recomendación
El módulo de autenticación es monolítico y difícil de modificar	Descomponer en microservicios
El sistema de clases en vivo no escala bien con más usuarios	Implementar un balanceador de carga
No hay redundancia en la base de datos	Implementar replicación en múltiples regiones

Modelos de Diseño Arquitectónico

Diseño Basado en Artefactos

El diseño basado en artefactos es un enfoque estructurado que se centra en la generación de documentos y modelos que representan distintas vistas del sistema.

Un **artefacto** en este contexto es cualquier resultado tangible del proceso de desarrollo, como:

- **Modelos de requisitos** (especificaciones funcionales, diagramas de casos de uso).
- **Modelos de diseño** (diagramas UML, diagramas de clases, diagramas de secuencia).
- **Documentación técnica** (manuales de usuario, documentación de arquitectura).
- **Código fuente y pruebas** (especificaciones de pruebas unitarias e integrales).

Este enfoque es utilizado en metodologías como el **Proceso Unificado de Desarrollo de Software (RUP)** y permite tener una trazabilidad clara desde los requisitos hasta la implementación.

Ejemplo

En el desarrollo de un sistema de gestión de inventario, los artefactos clave incluyen:

• Un **diagrama de clases** que modela las entidades **Producto, Almacén y Stock**.
• Un **diagrama de secuencia** que describe el flujo de actualización del stock al recibir un nuevo pedido.
• Un **documento de especificaciones** con los requisitos del sistema.

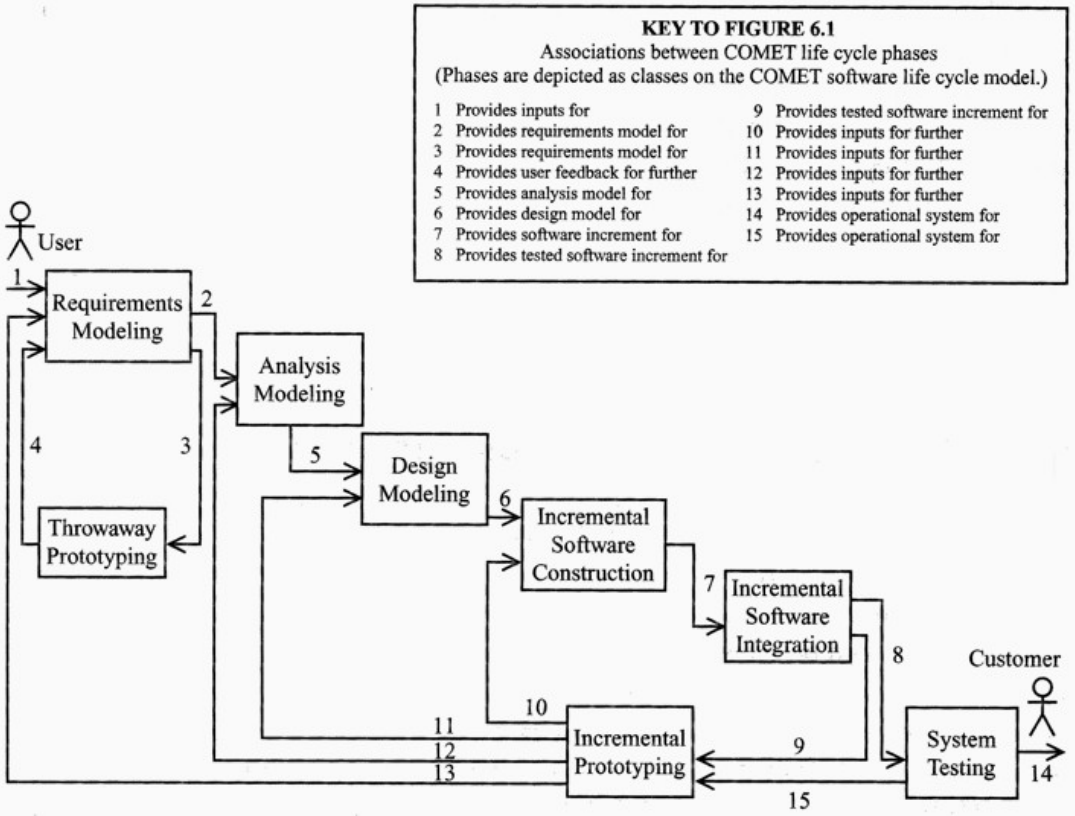
Evaluación del Diseño Basado en Artefactos

Método de Evaluación:

- **Revisión de Artefactos:** Se analizan los documentos generados (diagramas UML, especificaciones, modelos de datos, etc.).
- **Trazabilidad:** Se evalúa si los artefactos reflejan correctamente los requisitos y decisiones de diseño.
- **Consistencia y Completitud:** Se verifica que los artefactos sean coherentes entre sí y que no falten elementos clave.

Criterios de Evaluación:

- Claridad y precisión.
- Cohesión y acoplamiento.
- Documentación y mantenimiento.



Diseño Basado en Casos de Uso

El diseño basado en casos de uso parte del concepto de que el sistema debe ser modelado desde la perspectiva de los usuarios y las interacciones que tendrán con él.

En este enfoque se identifican los **casos de uso**, que representan escenarios funcionales específicos.

Un caso de uso describe:

- **Actor(es)**: Quiénes interactúan con el sistema.

- **Flujo principal**: Pasos normales que sigue el usuario para completar una tarea.

- **Flujos alternativos y excepciones**: Qué sucede si hay errores o variantes en el proceso.

Este enfoque es ampliamente utilizado en **ingeniería de requisitos** y en la modelación con UML, asegurando que el diseño refleje las expectativas del usuario antes de definir los detalles técnicos.

Ejemplo

Para una aplicación de comercio electrónico, un caso de uso podría ser **"Comprar un Producto"**, donde:

- **Actor**: Cliente.
- **Flujo principal**:
    1. El cliente selecciona un producto.
    2. Añade el producto al carrito.
    3. Ingresa los datos de pago y dirección de envío.
    4. Confirma la compra.
- **Excepciones**:
    - Si el pago falla, se solicita otro método de pago.
    - Si no hay stock, se notifica al usuario.

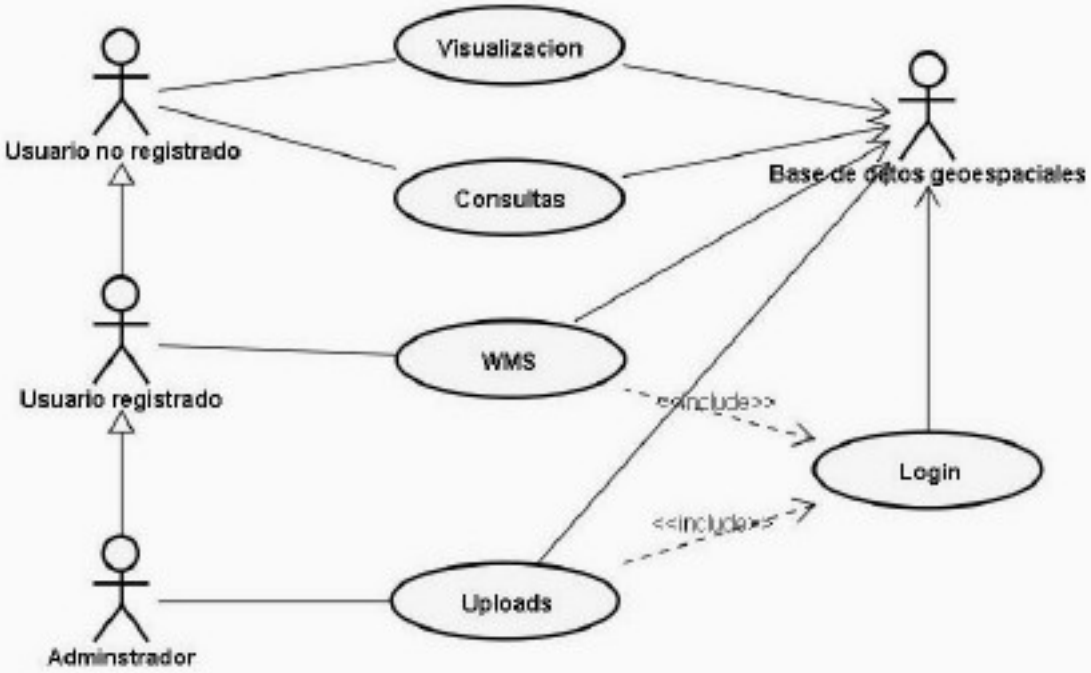
Evaluación del Diseño Basado en Casos de Uso

Método de Evaluación:

- **Revisión de Casos de Uso:** Se analiza si los casos de uso capturan correctamente los requisitos funcionales.
- **Validación con Usuarios:** Se hacen entrevistas y pruebas para verificar que los escenarios reflejan las necesidades reales.
- **Simulación de Escenarios:** Se ejecutan pruebas de recorrido sobre los casos de uso para identificar ambigüedades o inconsistencias.

Criterios de Evaluación:

- Cobertura de requisitos.
- Claridad y granularidad.
- Flujo y excepciones.



Diseño Basado en Modelos de Dominio

El diseño basado en modelos de dominio se centra en representar los conceptos y reglas del mundo real dentro del software.

Es un enfoque que busca capturar la esencia del problema a resolver mediante la identificación de entidades, relaciones y comportamientos.

Este enfoque es característico de **Domain-Driven Design (DDD)**

Ejemplo:

En un sistema de banca en línea, el modelo de dominio incluiría:

- **Entidad Cliente** con atributos como **nombre, dirección y cuentas bancarias asociadas**.
- **Entidad Cuenta Bancaria** con atributos como **saldo, tipo de cuenta y transacciones**.
- **Value Object Moneda**, que define el tipo de divisa sin identidad propia.

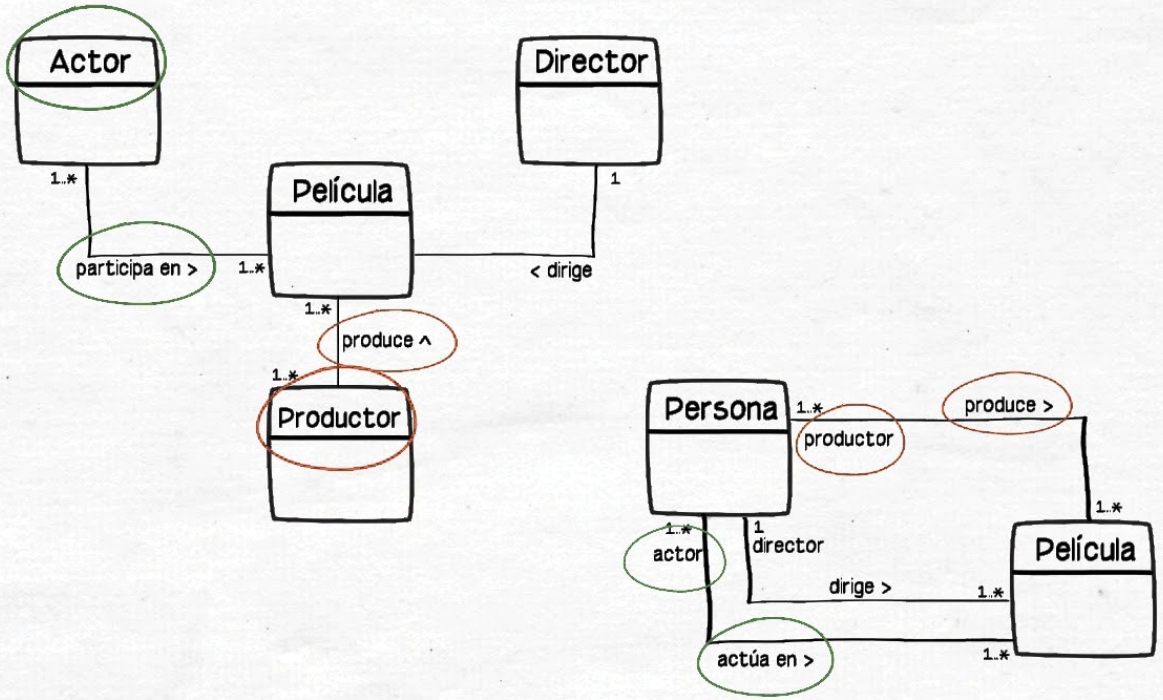
Evaluación del Diseño Basado en Modelos de Dominio

Método de Evaluación:

• **Validación con Expertos del Dominio:** Se presentan los modelos a expertos en el área del problema para verificar su precisión.
• **Pruebas de Integridad del Modelo:** Se revisan relaciones entre entidades, atributos y restricciones para garantizar consistencia.
• **Evaluación del Acoplamiento y Cohesión:** Se analiza si las clases y entidades reflejan correctamente la estructura del dominio y minimizan dependencias innecesarias.

Criterios de Evaluación:

- Representación del dominio.
- Escalabilidad y extensibilidad.
- Mantenimiento y claridad.

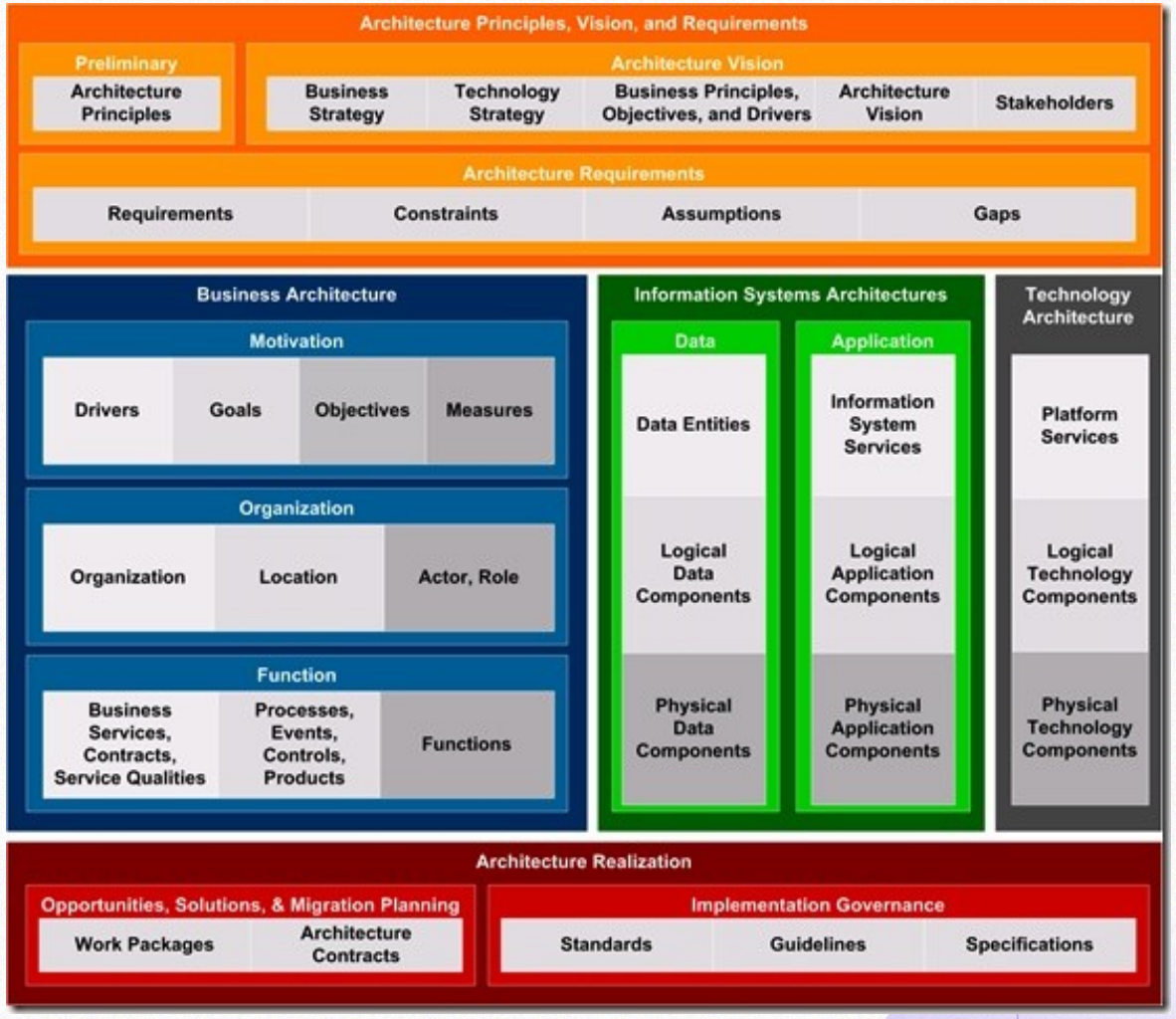


Meta-Modelo de Diseño Arquitectónico

Meta-Modelo de Diseño Arquitectónico

El **Meta-Modelo de Diseño Arquitectónico** es una abstracción que permite analizar y comparar distintos enfoques de diseño de arquitectura.

Su propósito es identificar los elementos clave en la construcción de arquitecturas de software y cómo estos interactúan.



Componentes del Meta-Modelo

El modelo está compuesto por **seis conceptos principales**, que representan las entidades involucradas en el proceso de diseño arquitectónico:

Cliente: Representa a los **stakeholders** (interesados en el desarrollo del sistema).

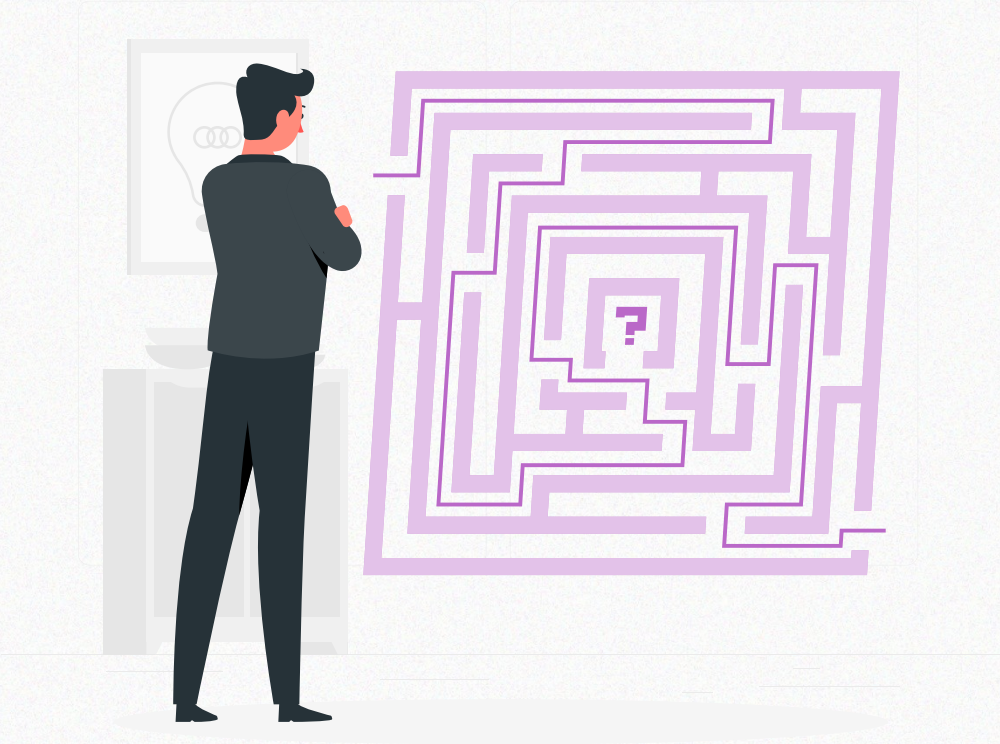
- Usuarios finales.
- Gerentes de negocio.
- Equipos de desarrollo.
- Mantenedores del sistema.

Rol en el modelo: Proporciona los **requisitos iniciales** que servirán como base para el diseño arquitectónico.

Conocimiento del Dominio: Es el conocimiento especializado necesario para diseñar la arquitectura.

Se divide en:

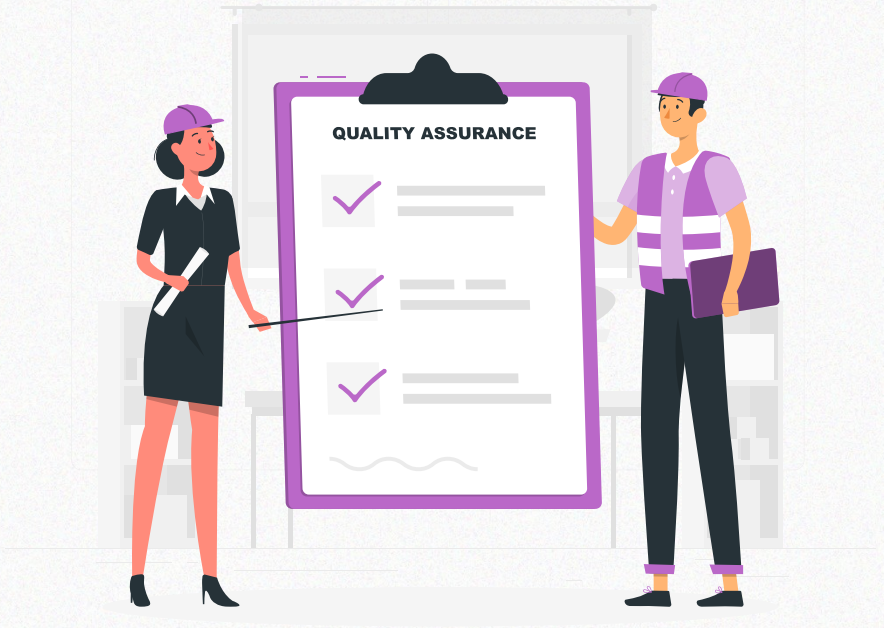
1. **Conocimiento del Problema**: Información sobre el dominio específico en el que opera el sistema.
2. **Conocimiento del Negocio**: Procesos empresariales y reglas operativas que el sistema debe cumplir.
3. **Conocimiento del Dominio de Solución**: Técnicas, herramientas y frameworks conocidos que pueden ser aplicados en el diseño.
4. **Conocimiento General**: Experiencia acumulada y heurísticas utilizadas en la ingeniería de software.



Especificación de Requisitos: Define **qué necesita el sistema** en términos de funcionalidad y atributos de calidad.

Puede incluir:

- Requisitos funcionales (lo que el sistema debe hacer).
- Requisitos no funcionales (rendimiento, seguridad, escalabilidad, etc.).



Relación con el Cliente y Conocimiento del Dominio: Los requisitos son definidos por los stakeholders y validados con base en el conocimiento del dominio.

Artefactos: Son los documentos o modelos generados durante el diseño de la arquitectura.

Ejemplos:

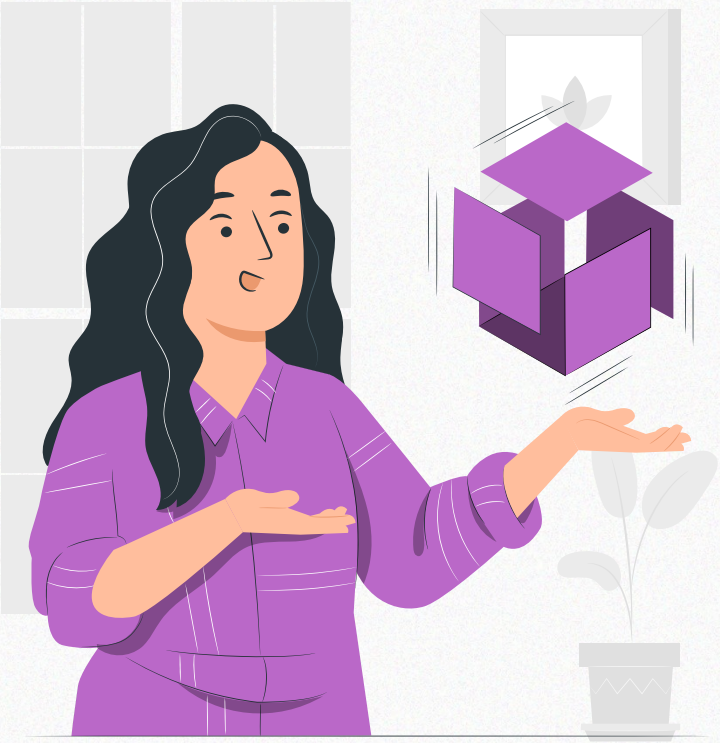
- Diagramas de clases.
- Diagramas de flujo de datos.
- Modelos de casos de uso.
- Modelos de dominio.

Función en el modelo: Sirven como base para generar las abstracciones arquitectónicas.

Abstracción de la Solución: Es la **representación conceptual** de la arquitectura basada en los artefactos y requisitos.

Define los principales **componentes del sistema** y sus interacciones. Puede expresarse en términos de:

- Módulos.
- Servicios.
- Microservicios.
- Capas del sistema.

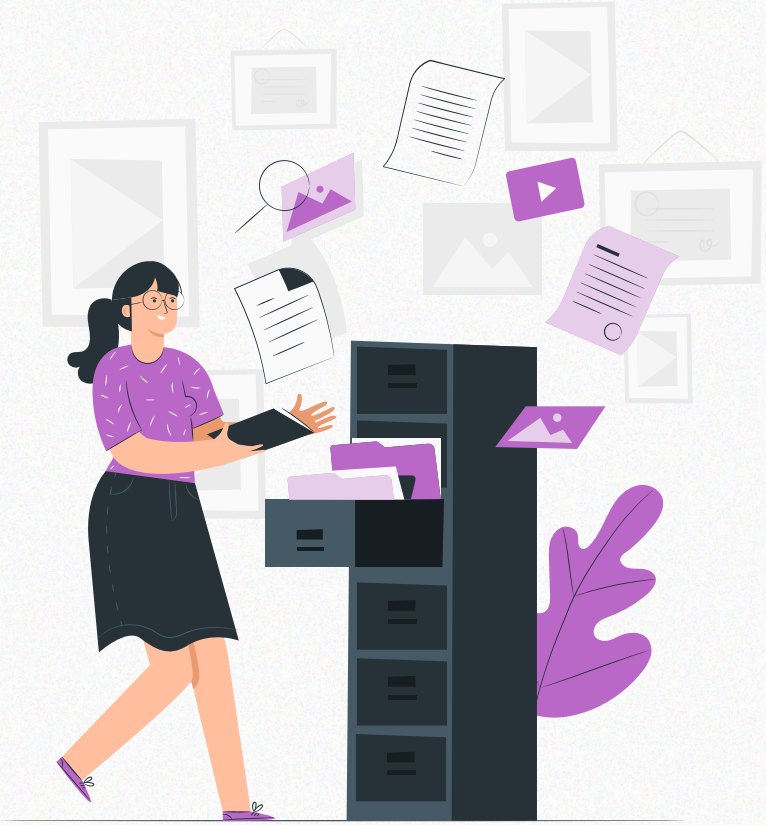


Descripción de la Arquitectura: Documento formal que describe la estructura del sistema.

Contiene:

- Componentes.
- Relaciones entre componentes.
- Reglas y restricciones arquitectónicas.

Uso en el modelo: Sirve como guía para la implementación y evaluación de la arquitectura.

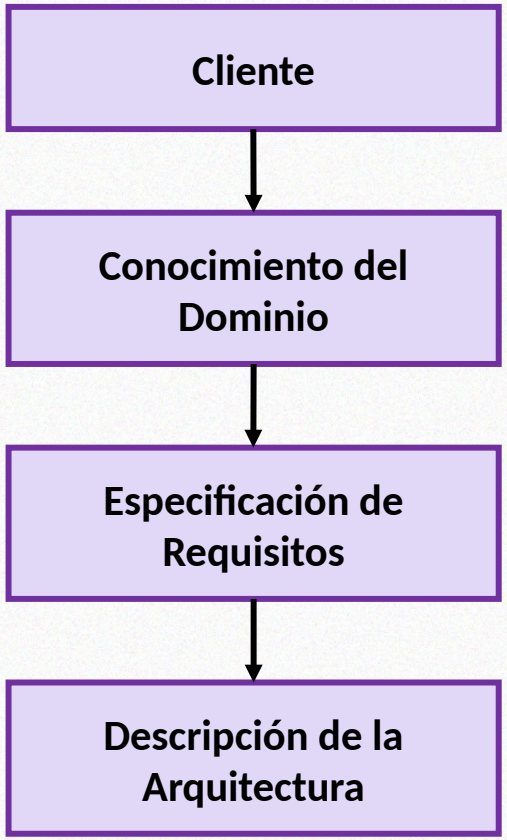


Relaciones Clave en el Meta-Modelo

El meta-modelo define tres asociaciones fundamentales que explican cómo los diferentes conceptos interactúan entre sí:

Captura de Requisitos

- El cliente define los requisitos con base en su conocimiento del negocio.
- Los arquitectos validan estos requisitos usando conocimiento previo del dominio.
- Se genera un documento formal con la especificación de requisitos.

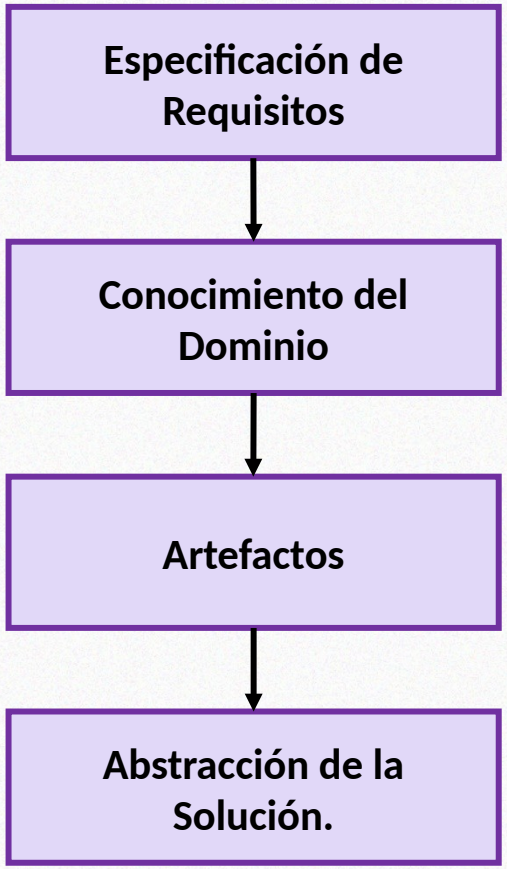


Relaciones Clave en el Meta-Modelo

El meta-modelo define tres asociaciones fundamentales que explican cómo los diferentes conceptos interactúan entre sí:

Extracción de Estructuras de Solución

- A partir de los requisitos y el conocimiento del dominio, se generan artefactos.
- Los artefactos se agrupan y analizan para formar abstracciones arquitectónicas.

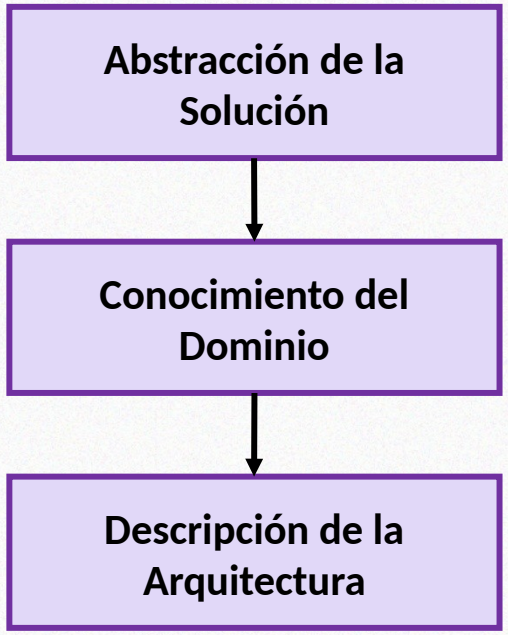


Relaciones Clave en el Meta-Modelo

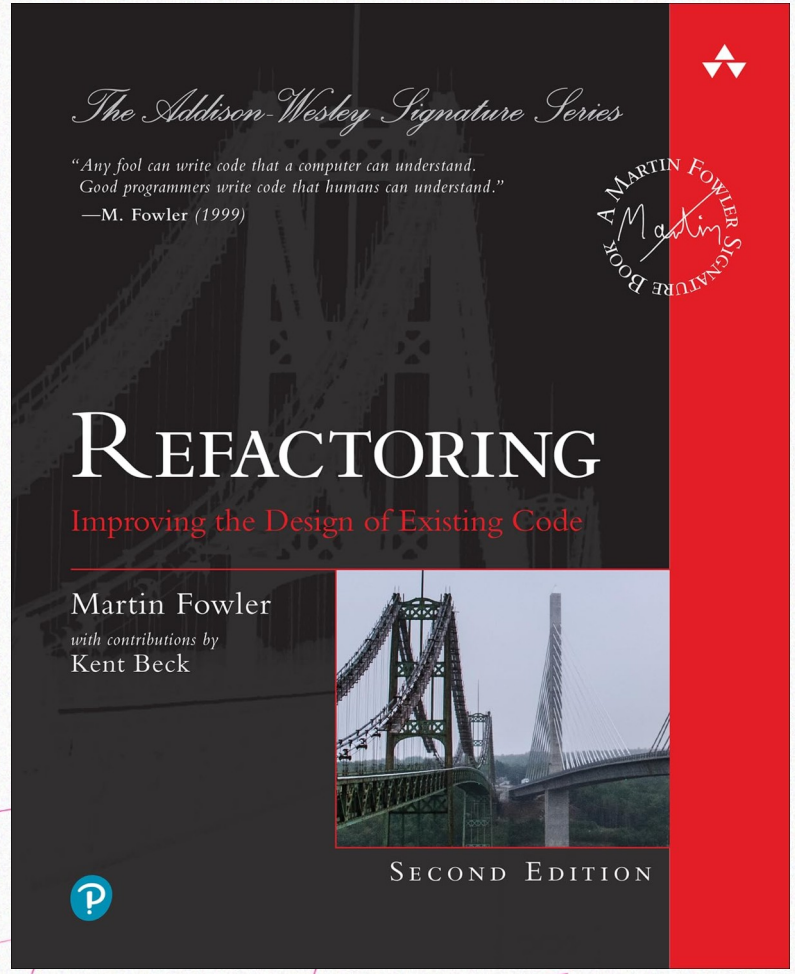
El meta-modelo define tres asociaciones fundamentales que explican cómo los diferentes conceptos interactúan entre sí:

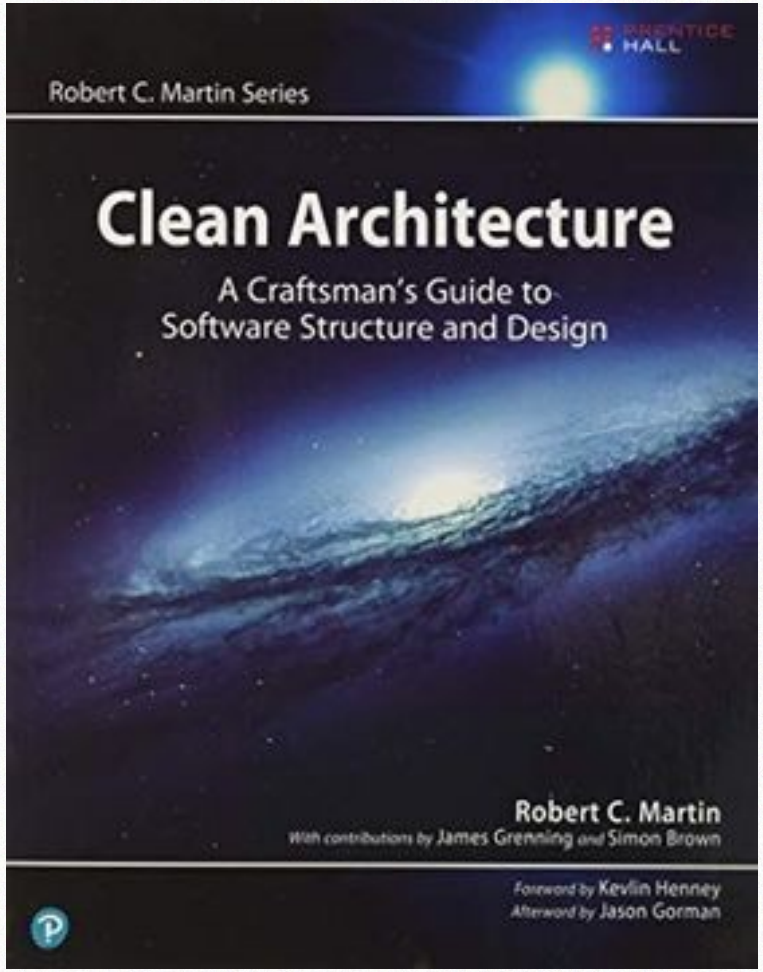
Especificación de la Arquitectura

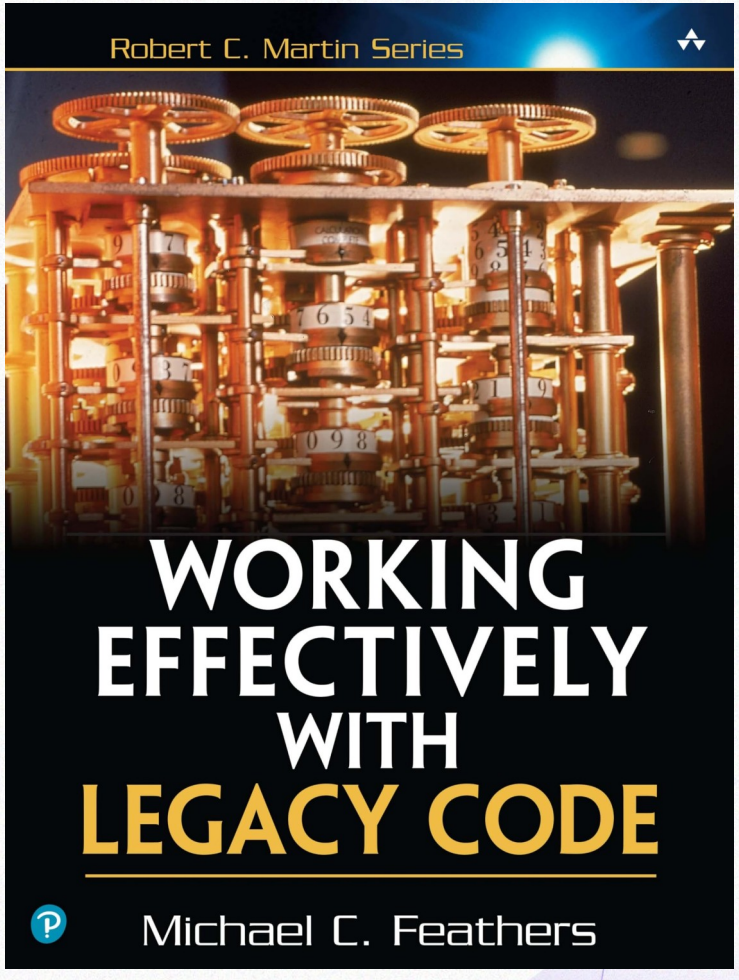
- Se formaliza la arquitectura mediante modelos y documentación.
- Se establecen principios de diseño basados en conocimientos previos y mejores prácticas.



Refactorización y mejora de la arquitectura existente


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition


Prentice Hall
Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to
Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman


Robert C. Martin Series
WORKING
EFFECTIVELY
WITH
LEGACY CODE
Michael C. Feathers

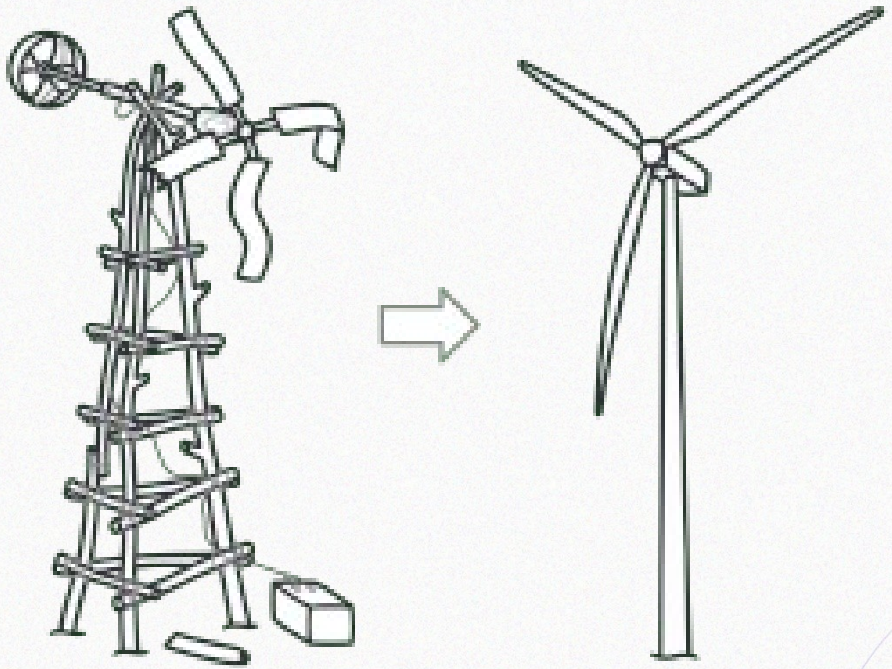
Introducción a la Refactorización Arquitectónica

El término **refactorización** fue popularizado por **Martin Fowler (1999)** en su libro *Refactoring: Improving the Design of Existing Code*, donde la define como:

> "*Un cambio en la estructura interna del software que mejora su diseño sin alterar su comportamiento observable.*"

¿Qué pasa cuando ampliamos este concepto a la arquitectura del software?

La **refactorización arquitectónica** es el proceso mediante el cual mejoramos la estructura general de un sistema, reorganizando sus componentes, desacoplando módulos y optimizando su escalabilidad sin cambiar el propósito fundamental del software.

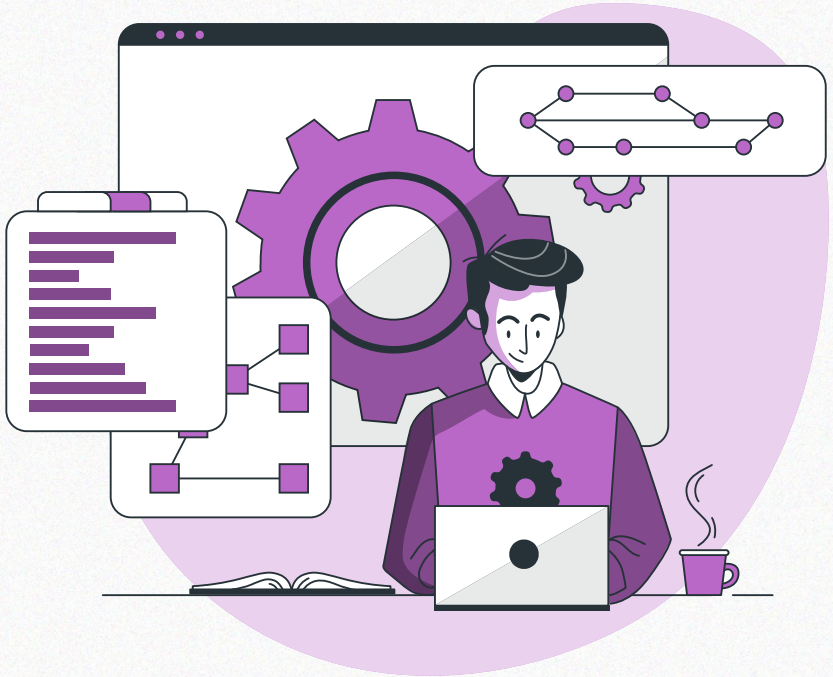


Ejemplo

Imagina que tienes una **aplicación monolítica** que ha crecido demasiado. Inicialmente, la arquitectura funcionaba bien, pero con el tiempo, la base de código se ha vuelto inmanejable:

- Cada nuevo cambio **afecta múltiples módulos**.
- Hay una fuerte **dependencia entre componentes**.
- La aplicación **no escala** correctamente con la demanda.

Aquí es donde entra la refactorización arquitectónica: podemos **separar gradualmente los módulos en una arquitectura más flexible, como una basada en microservicios o en capas bien definidas**.



¿Cuándo la refactorización deja de ser solo sobre código y empieza a ser arquitectónica?

Refactorización de código	Extraer una función de 100 líneas en funciones más pequeñas.
Refactorización arquitectónica	Convertir una base de datos monolítica en múltiples bases de datos especializadas.

¿Por qué creen que los sistemas tienden a deteriorarse con el tiempo?

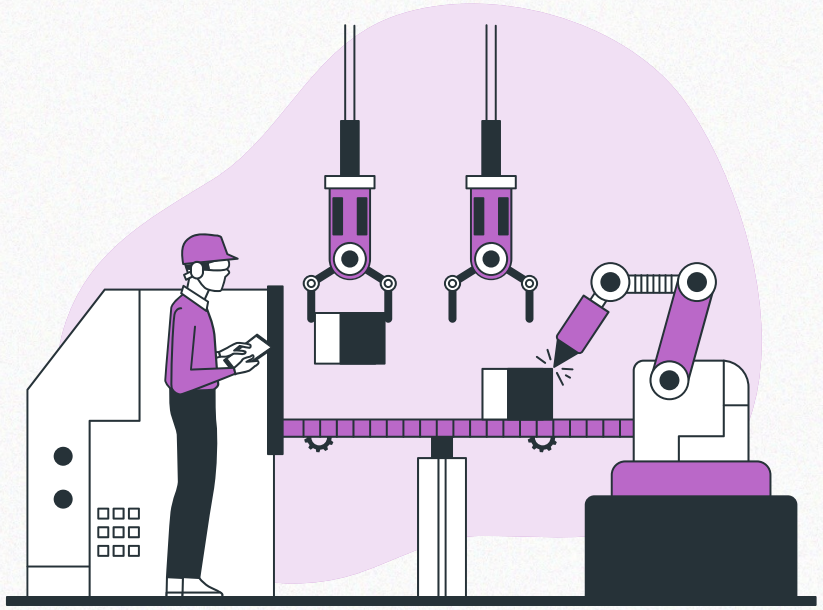
Razones para Refactorizar una Arquitectura Existente

A medida que un sistema crece, es inevitable que su arquitectura se vuelva menos eficiente.

Esto se debe a varios factores, como cambios en los requisitos de negocio, evolución tecnológica y crecimiento de la base de usuarios.

Según **Michael Feathers (2004)**, la acumulación de **deuda técnica** es una de las principales razones por las que se debe considerar la refactorización. La **deuda técnica** se define como:

> *"El costo adicional requerido para modificar un sistema debido a decisiones previas que priorizaron la entrega rápida sobre la calidad del diseño."*



Algunas razones para considerar refactorizar la arquitectura

1. **Deuda técnica acumulada:**
- El sistema es difícil de modificar porque **está altamente acoplado**.
- Falta de patrones de diseño adecuados.

2. **Problemas de escalabilidad:**
- La arquitectura actual no soporta el crecimiento del tráfico o los datos.
- Dependencias innecesarias generan **cuellos de botella**.

3. **Dificultades en la integración con nuevos sistemas:**
- La arquitectura impide la comunicación con APIs modernas.
- Se requiere agregar nuevas funcionalidades y la estructura existente no lo permite fácilmente.

4. **Alto costo de mantenimiento:**
- Pequeños cambios requieren **grandes esfuerzos** de desarrollo.
- Hay **duplicación de código y falta de modularidad**.

Ejemplo

Imagina un ecommerce que comenzó como un pequeño sitio web, pero con el tiempo ha crecido en funcionalidades.

Ahora maneja catálogos de productos, pagos en línea y logística de envíos.

Sin embargo, sigue siendo un sistema monolítico. Cada vez que los desarrolladores quieren hacer un cambio, deben revisar cientos de líneas de código no modularizado.

Refactorización vs. Reescritura: ¿Cuál elegir?

Refactorizar cuando:

- El sistema sigue cumpliendo su propósito, pero es difícil de modificar.
- Podemos dividir la refactorización en **etapas progresivas**.
- La migración completa es demasiado costosa y arriesgada.

Reescribir cuando:

- El código es inmantenible y no tiene documentación.
- No existen expertos en la tecnología utilizada.
- Se necesita un rediseño fundamental del modelo de negocio.

Caso de Estudio

OptiLogistics es una empresa de logística que comenzó hace cinco años con una aplicación monolítica para gestionar envíos, seguimiento de paquetes y gestión de clientes. El sistema funcionó bien al principio, pero a medida que la empresa creció y aumentó el volumen de usuarios, los problemas comenzaron a acumularse.

- **Tiempo de respuesta lento:** A medida que se agregan más clientes y pedidos, las consultas a la base de datos central tardan demasiado.
- **Dificultades en la integración:** La empresa quiere conectar su sistema con API de proveedores externos (como DHL, FedEx y UPS), pero la estructura actual lo hace complicado.
- **Dependencias entre módulos:** Cada cambio en la gestión de clientes afecta la parte de facturación, inventario y reportes.
- **Código inmanejable:** No hay una clara separación entre capas; la lógica de negocio, la base de datos y la interfaz están fuertemente acopladas.
- **Tiempo de inactividad elevado:** Cada actualización del sistema requiere detener todos los servicios, ya que el monolito no permite despliegues parciales.
- **Dificultad para expandirse a nuevos mercados:** La empresa quiere expandirse a otros países, pero la arquitectura actual **no soporta múltiples zonas horarias ni configuraciones regionales**.

Caso de Estudio

Si fueras el arquitecto de software de OptiLogistics

¿Qué harías para mejorar la situación?

Métodos para Identificar Problemas en una Arquitectura Existente

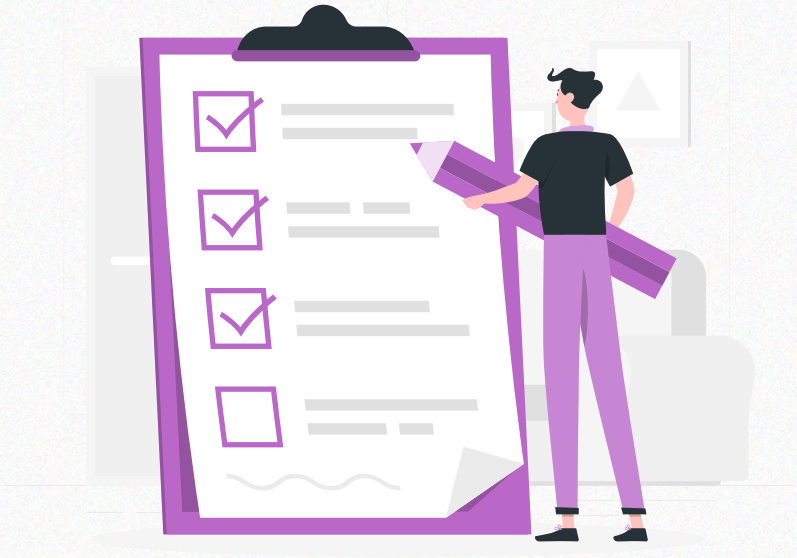
Modelo de las “4 S” de Problemas Arquitectónicos

Scalability (Escalabilidad): El sistema no maneja bien grandes volúmenes de datos o usuarios.

Security (Seguridad): Vulnerabilidades o exposición de datos.

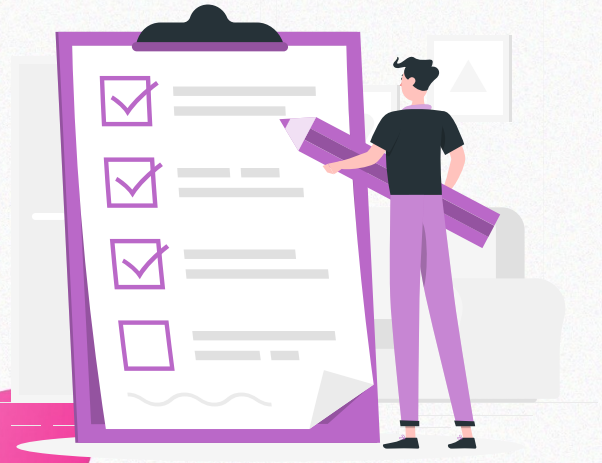
Stability (Estabilidad): El sistema es propenso a fallas o interrupciones.

Simplicty (Simplicidad): Código o arquitectura demasiado compleja.



ATAM (Architecture Tradeoff Analysis Method)

ATAM es un método formal desarrollado por el **Software Engineering Institute (SEI)** de Carnegie Mellon para evaluar arquitecturas de software. Se centra en **cómo diferentes decisiones arquitectónicas afectan múltiples atributos de calidad** (rendimiento, seguridad, escalabilidad, etc.).



Identificar los objetivos de negocio.
• Escalabilidad: soportar más clientes sin reducir el rendimiento.
• Integrabilidad: permitir conexión con API de terceros.

Analizar decisiones arquitectónicas actuales.
• Uso de una sola base de datos → No escalable.
• Arquitectura monolítica → Difícil de actualizar sin afectar todo el sistema.

Identificar riesgos y compromisos.
• **Riesgo:** Si se divide el monolito, ¿cómo garantizar la integridad de los datos?
• **Compromiso:** Una arquitectura desacoplada mejora la escalabilidad pero aumenta la complejidad.

Modelo de Evaluación Técnica-Business Impact Matrix

Este enfoque permite clasificar problemas **según su impacto técnico y de negocio**, ayudando a decidir **qué problemas resolver primero**.

IMPACTO TECNICO

Prioritario	**Critico**
Baja Prioridad	**Importante**

IMPACTO EN EL NEGOCIO

ARID (Architecture Refactoring Identification and Decision)

Este es un enfoque estructurado para **identificar y decidir refactorizaciones arquitectónicas**.

Se basa en tres pasos:

- **Identificación del problema:** ¿Dónde están los puntos débiles del sistema?
- **Clasificación del problema:** ¿Es un problema de rendimiento, escalabilidad, acoplamiento, etc.?
- **Decisión sobre la solución:** ¿Se puede refactorizar o es mejor reescribir esa parte del sistema?

Caso de Estudio

OptiLogistics es una empresa de logística que comenzó hace cinco años con una aplicación monolítica para gestionar envíos, seguimiento de paquetes y gestión de clientes. El sistema funcionó bien al principio, pero a medida que la empresa creció y aumentó el volumen de usuarios, los problemas comenzaron a acumularse.

- **Tiempo de respuesta lento:** A medida que se agregan más clientes y pedidos, las consultas a la base de datos central tardan demasiado.
- **Dificultades en la integración:** La empresa quiere conectar su sistema con API de proveedores externos (como DHL, FedEx y UPS), pero la estructura actual lo hace complicado.
- **Dependencias entre módulos:** Cada cambio en la gestión de clientes afecta la parte de facturación, inventario y reportes.
- **Código inmanejable:** No hay una clara separación entre capas; la lógica de negocio, la base de datos y la interfaz están fuertemente acopladas.
- **Tiempo de inactividad elevado:** Cada actualización del sistema requiere detener todos los servicios, ya que el monolito no permite despliegues parciales.
- **Dificultad para expandirse a nuevos mercados:** La empresa quiere expandirse a otros países, pero la arquitectura actual **no soporta múltiples zonas horarias ni configuraciones regionales**.

Fase 1: Identificación de Problemas

Esta fase consiste en **detectar los puntos críticos** del sistema y documentar sus síntomas. Se pueden utilizar herramientas y metodologías de análisis para encontrar problemas ocultos.

- **Análisis de Métricas de Código y Arquitectura:** Uso de herramientas como SonarQube o ArchUnit.
- **Evaluaciones de Escalabilidad:** Pruebas de carga para detectar cuellos de botella.
- **Entrevistas con el Equipo de Desarrollo:** Preguntar a los ingenieros sobre dificultades frecuentes.
- **Análisis de Logs y Monitoreo:** Identificar errores recurrentes o tiempos de respuesta elevados.

¿Cuáles son los principales obstáculos que enfrenta nuestra arquitectura?”

Fase 1: Identificación de Problemas

Síntoma observado	Posible causa técnica
🔴 **Tiempo de respuesta lento en consultas de envíos**	Base de datos centralizada sin optimización de índices ni particionamiento.
🔴 **Fuerte dependencia entre módulos de negocio**	Código monolítico sin separación clara de responsabilidades.
🔴 **Cada actualización del sistema requiere detener todo el servicio**	No hay soporte para despliegues parciales ni arquitectura modular.
🟠 **Dificultad para integrar nuevos proveedores logísticos (FedEx, DHL, etc.)**	No hay un API Gateway, lo que genera acoplamiento con proveedores externos.
🟠 **No hay registros de monitoreo centralizados**	Falta de logging y trazabilidad en los servicios.

Evidencia recopilada:

- **Pruebas de carga** → Las consultas a la base de datos muestran tiempos de respuesta inaceptables.

- **Reportes de incidencias** → El equipo de desarrollo indica que **cada cambio afecta múltiples módulos**, lo que genera incertidumbre y retrasos en entregas.

- **Monitoreo de logs** → Se detectaron **errores recurrentes de timeout en el acceso a la base de datos**.

Fase 2: Clasificación y Priorización de Problemas

Para evitar caer en refactorizaciones innecesarias, los problemas se **clasifican y priorizan** utilizando criterios como impacto técnico, impacto en el negocio y esfuerzo requerido para resolverlos.

Quick Wins	**Proyectos Principales**
Tareas Menores	**Descartar o Postergar**

IMPACTO

ESFUERZO

¿Cuáles de estos problemas tienen el mayor impacto en el negocio?

¿Cuánto esfuerzo se necesita para resolverlos?

¿Cuáles deben resolverse primero?

Fase 2: Clasificación y Priorización de Problemas

IMPACTO

Quick Wins • Optimización de Base de Datos • Indexación de consultas críticas • Implementación de Caching con Redis	**Proyectos Principales** • Desacoplamiento del módulo de Gestión de Envíos • Creación de un API Gateway • Implementación de comunicación asíncrona con RabbitMQ
Tareas Menores • Mejoras en logging y monitoreo - Documentación de la API interna	**Descartar o Postergar** • Rediseño de UI/UX - Migración completa a microservicios

ESFUERZO

Fase 3: Decisión y Estrategia de Refactorización

Con los problemas priorizados, se definen estrategias de refactorización según el contexto del sistema:

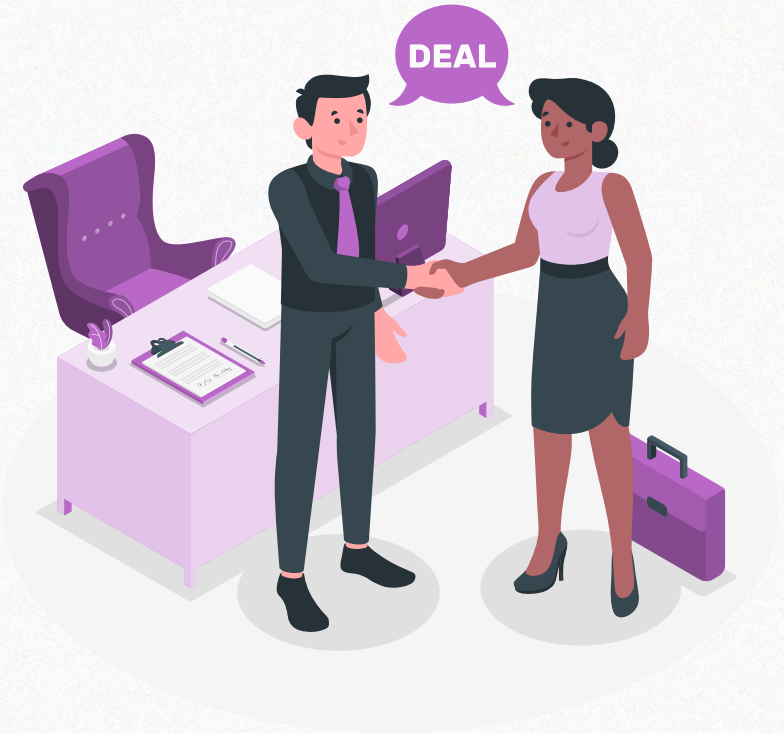
- **Refactorización progresiva:** Si los cambios pueden implementarse en pequeñas iteraciones sin interrumpir el servicio.
- **Migración paralela:** Si es necesario crear nuevos módulos antes de eliminar los antiguos.
- **Reescritura total:** Si la arquitectura existente no puede adaptarse a los nuevos requerimientos.

¿Cómo implementamos estos cambios sin interrumpir la operación del sistema?

¿Debemos refactorizar progresivamente o hacer una migración completa?

Fase 3: Decisión y Estrategia de Refactorización

Estrategia seleccionada: Aplicar el patrón **“Strangler Fig”.** En lugar de hacer una reescritura completa, migraremos gradualmente partes del monolito hacia una arquitectura más modular.



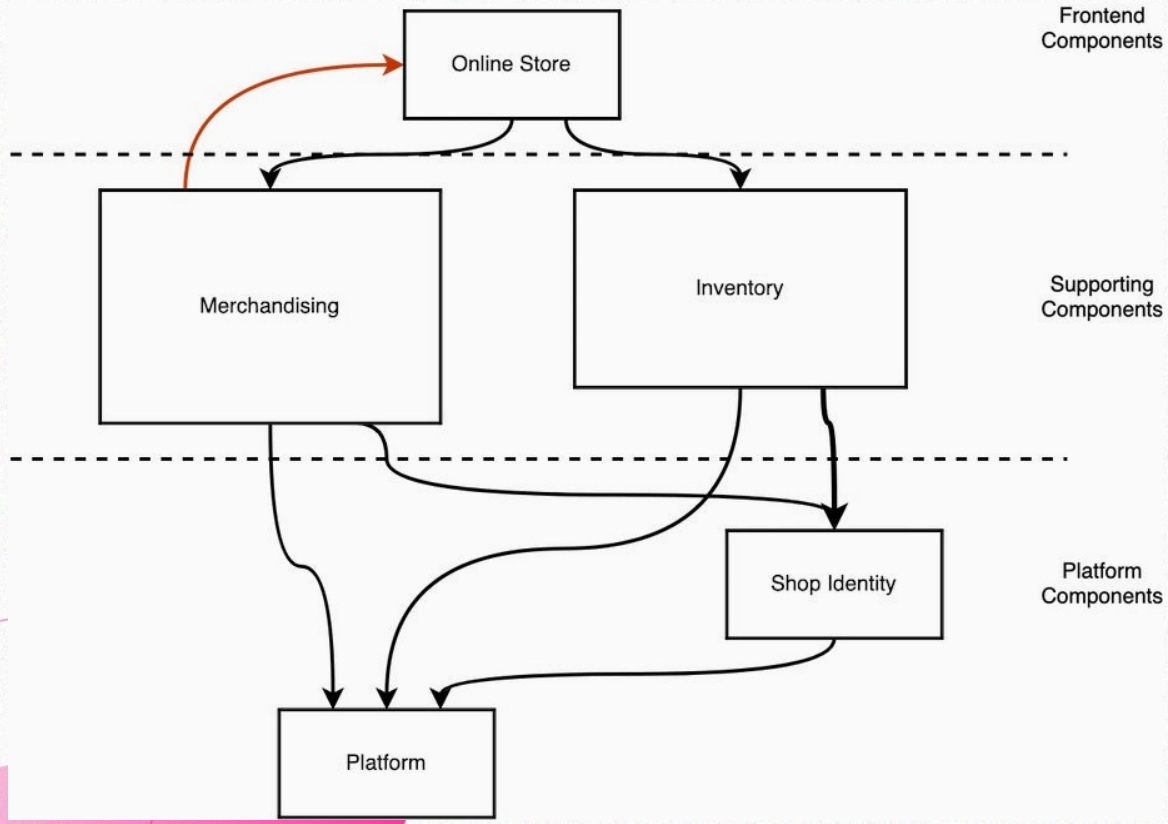
Plan de Refactorización Progresiva (Roadmap en 3 fases):

- **Fase 1: Optimización de la Base de Datos**
- **Fase 2: Desacoplamiento de Servicios Críticos**
- **Fase 3: Implementación de API Gateway**

Tecnicas comunes para Modernizar software herdado

Deconstrucción del Monolito

Es el proceso de dividir un **sistema monolítico en módulos independientes**, permitiendo mayor escalabilidad y flexibilidad.



Estrategia para la deconstrucción

Paso 1: Identificar los módulos clave
• Separar la lógica en **módulos funcionales** (Ej. Autenticación, Pagos, Inventario).
• Usar métricas como **cambio frecuente en el código** para definir prioridades.

Paso 2: Exponer módulos mediante APIs
• Crear una **capa de abstracción** que permita el acceso a la lógica de negocio sin modificar la implementación interna.
• Implementar **APIs REST o GraphQL** para cada funcionalidad.

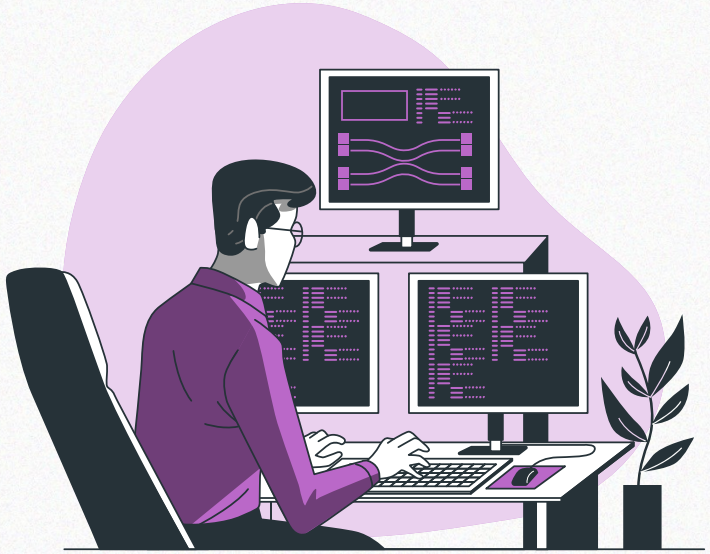
Paso 3: Separar la base de datos
• Adoptar una **arquitectura de** bases de datos independientes o mediante **CQRS**.

Reescritura Incremental

Es el proceso de **reemplazar gradualmente partes del sistema sin interrumpir el funcionamiento**.

Estrategia para la deconstrucción

- **Identificar componentes acoplados** y extraer su lógica a módulos más pequeños.
- **Usar Feature Toggles** para alternar entre la implementación antigua y la nueva.
- **Probar cada iteración** con datos reales antes de sustituir completamente la funcionalidad antigua.



Reescritura Incremental

Es el proceso de **reemplazar gradualmente partes del sistema sin interrumpir el funcionamiento**.

```
class OrderService:
    def create_order(self, user, items):
        if not self.validate_stock(items):
            return "Stock insuficiente"
        order = Order(user=user, items=items)
        self.save_to_database(order)
        return order
```

```
class OrderValidator:
    def validate_stock(self, items):
        # Separación de lógica de validación
        return all(item.stock > 0 for item in iter
```

```
class OrderRepository:
    def save(self, order):
        # Separación de persistencia
        database.save(order)
```

```
class OrderService:
    def create_order(self, user, items):
        validator = OrderValidator()
        repository = OrderRepository()

        if not validator.validate_stock(items):
            return "Stock insuficiente"

        order = Order(user=user, items=items)
        repository.save(order)
        return order
```

Segmentación Progresiva con APIs

La segmentación progresiva con APIs es una estrategia de modernización que permite dividir un sistema monolítico en módulos independientes sin una reescritura completa ni una interrupción drástica en su funcionamiento

Estrategia

- Identificar el módulo a segmentar.
- Exponer una API para ese módulo sin modificar el código original.
- Redirigir las solicitudes a la nueva API de manera controlada.
- Optimizar y migrar la implementación interna.
- Eliminar el código original del monolito cuando la API esté estable.

Segmentación Progresiva con APIs

La segmentación progresiva con APIs es una estrategia de modernización que permite dividir un sistema monolítico en módulos independientes sin una reescritura completa ni una interrupción drástica en su funcionamiento

```
class ProductService:
    def get_product_details(self, product_id):
        return database.query("SELECT * FROM products WHERE id = ?", product_id)
```

```
from flask import Flask, jsonify
import database

app = Flask(__name__)

@app.route("/products/<id>", methods=["GET"])
def get_product_details(id):
    product = database.query("SELECT * FROM products WHERE id = ?", id)
    return jsonify(product)

if __name__ == "__main__":
    app.run(port=5001)
```

Segmentación Progresiva con APIs

La segmentación progresiva con APIs es una estrategia de modernización que permite dividir un sistema monolítico en módulos independientes sin una reescritura completa ni una interrupción drástica en su funcionamiento

Estrategia

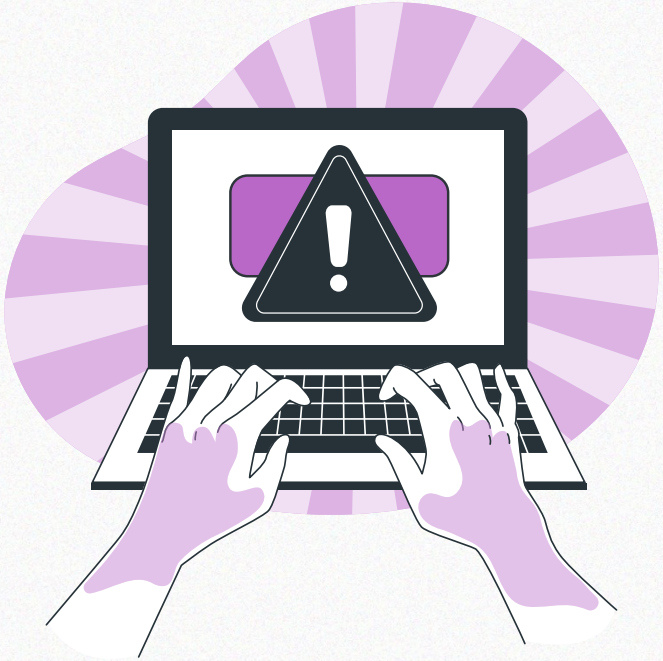
- Identificar el módulo a segmentar.
- Exponer una API para ese módulo sin modificar el código original.
- Redirigir las solicitudes a la nueva API de manera controlada.
- Optimizar y migrar la implementación interna.
- Eliminar el código original del monolito cuando la API esté estable.

Cómo preparar un sistema para una migración sin riesgos

¿Por qué una migración puede ser riesgosa?

Las migraciones fallidas pueden generar:

- **Interrupciones en el servicio:** Si algo falla, la empresa puede perder clientes.
- **Pérdida de datos:** Un error en la transición puede ser irreversible.
- **Regresión en funcionalidades:** Si no se prueban correctamente los módulos migrados, pueden romper el flujo del negocio.
- **Impacto en el rendimiento:** Nuevas arquitecturas pueden no estar optimizadas desde el primer momento.

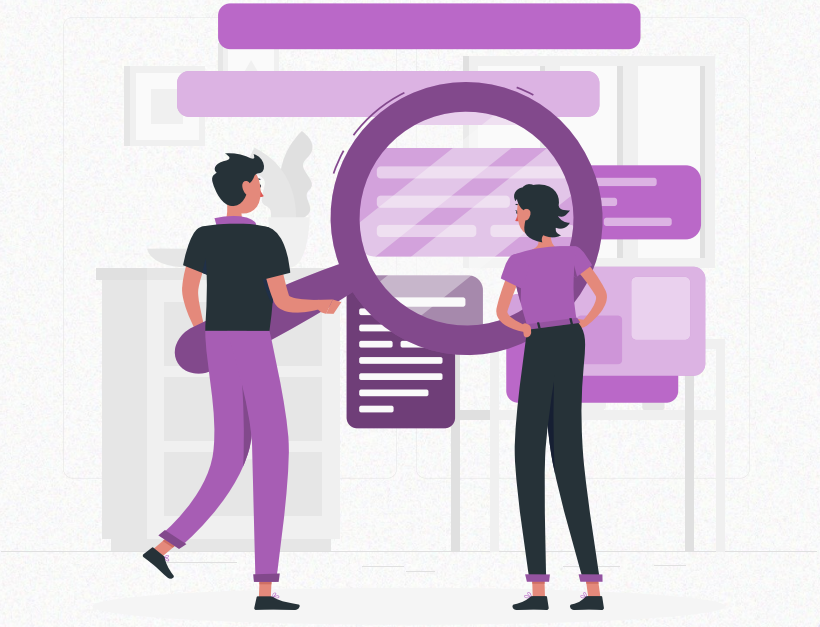


Estrategias para una Migración Segura

Análisis del sistema actual

Antes de migrar cualquier sistema, debemos entender **qué estamos migrando**.

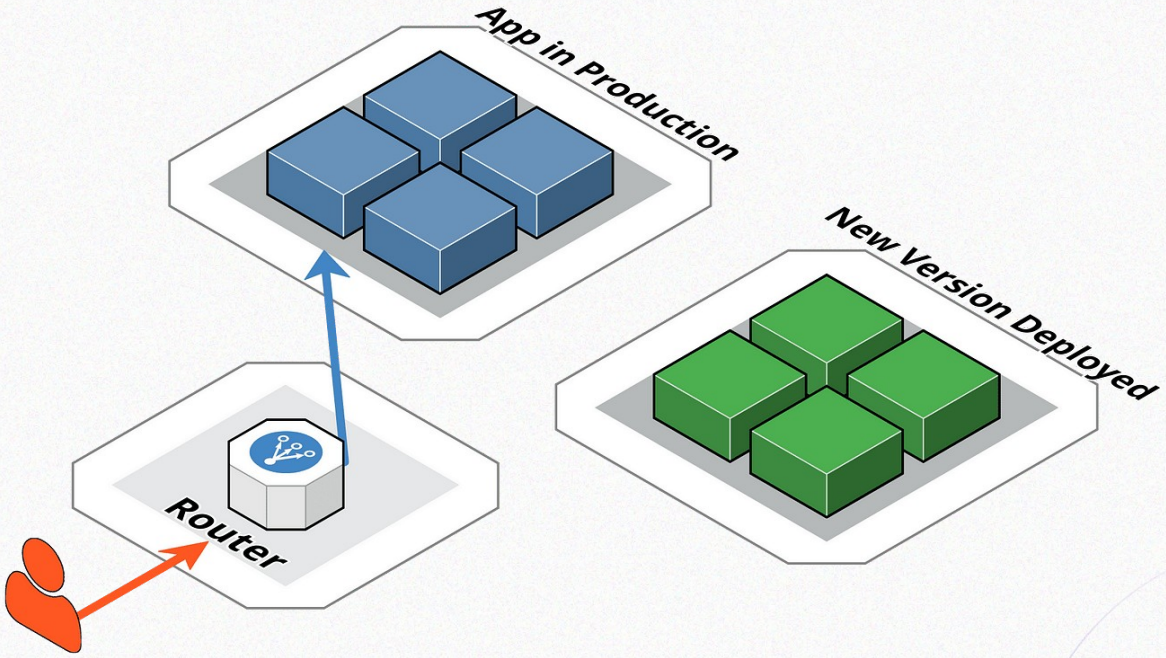
- ¿Cuáles son los módulos más críticos?
- ¿Qué partes del sistema son más utilizadas por los usuarios?
- ¿Cuáles son los puntos de mayor deuda técnica?
- ¿Cuáles son las dependencias entre módulos?
- ¿Cuáles son los tiempos de respuesta y carga de cada parte del sistema?



Estrategias para una Migración Segura

Blue/Green Deployment:

Se despliega la nueva versión en paralelo sin eliminar la anterior, cuando todo está funcionando bien, el tráfico se redirige a la nueva versión.



Estrategias para una Migración Segura

Feature Toggles:

Permiten activar y desactivar partes del nuevo código sin hacer un despliegue completo.

Se pueden probar nuevas funcionalidades en producción sin que todos los usuarios las vean.

Si algo sale mal, **basta con cambiar la variable sin tocar el código**.

```
FEATURE_FLAGS = {"new_checkout": False}

def checkout():
    if FEATURE_FLAGS["new_checkout"]:
        return new_checkout_system()
    else:
        return old_checkout_system()
```

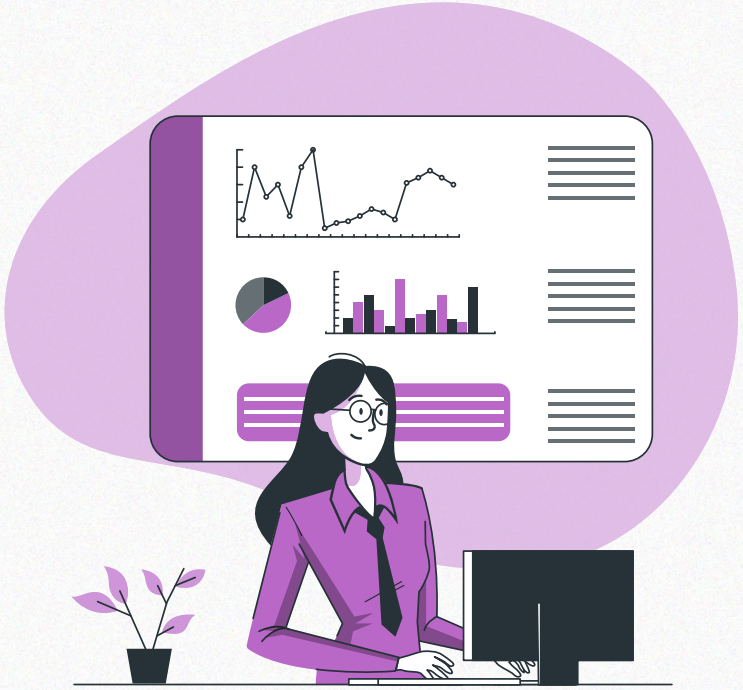
Monitoreo y Pruebas Antes y Después de la Migración

Pruebas antes de la migración:

- **Pruebas de carga:** Simular tráfico real para evaluar rendimiento.
- **Pruebas de regresión:** Verificar que la nueva implementación no rompe funcionalidades existentes.
- **Pruebas de integración:** Asegurar que el nuevo sistema funciona con los módulos antiguos.

Pruebas después de la migración:

- **Monitoreo de métricas clave:** Tiempo de respuesta, errores HTTP, tasa de conversión.
- **Pruebas A/B:** Redirigir tráfico de usuarios reales a la nueva versión para comparar su rendimiento.

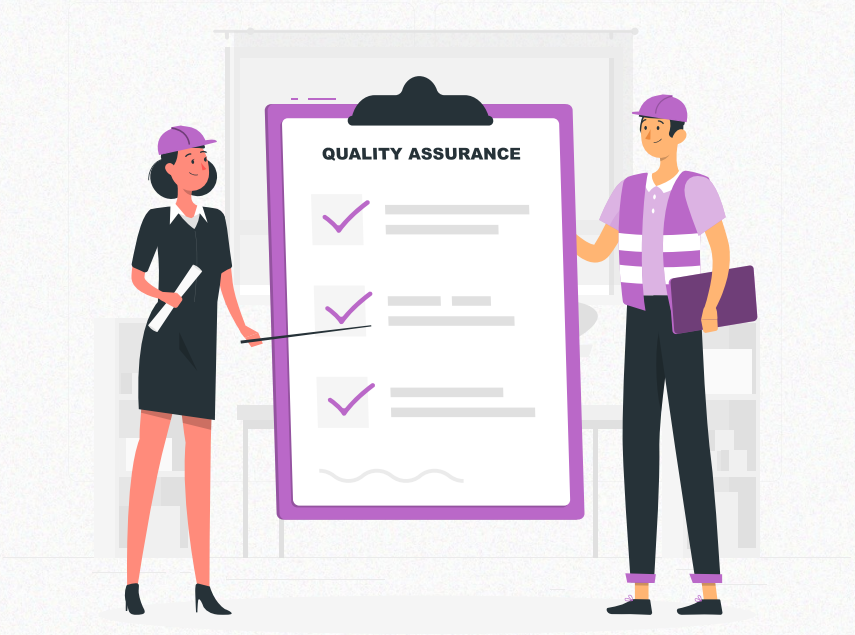


Refactorización Guiada por Pruebas (TDD en Arquitectura)

Usar **pruebas automatizadas** para guiar la refactorización y evitar errores en producción.

Asegura que los cambios no rompan funcionalidades existentes.

Permite reestructurar la arquitectura sin afectar el comportamiento.

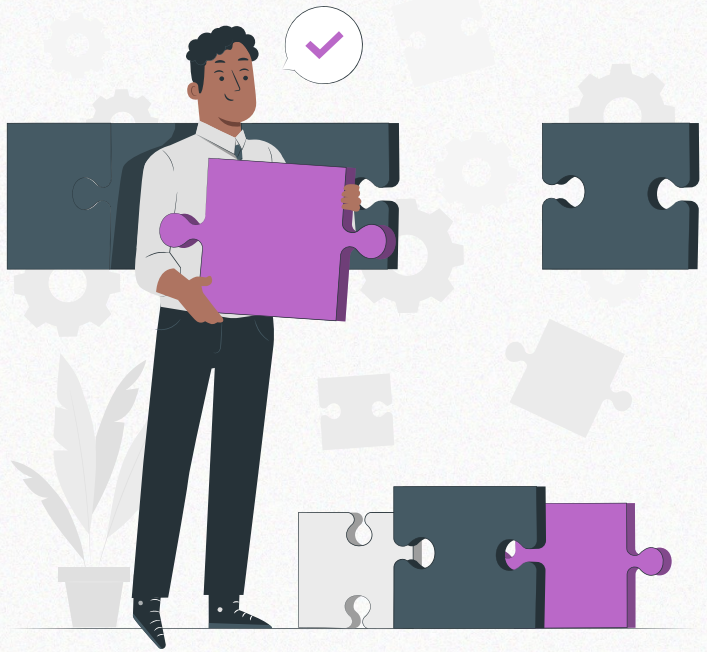


Principales Patrones Arquitectónicos Evolutivos

Patrones de Evolución Arquitectónica

La evolución del software es un proceso inevitable en cualquier sistema en producción. A medida que las necesidades del negocio cambian y las tecnologías avanzan, las arquitecturas deben adaptarse para seguir siendo eficientes y mantenibles. Sin embargo, este proceso no siempre ocurre de manera controlada, lo que puede llevar a la degradación de la arquitectura, el aumento del *debt técnico* y la pérdida de flexibilidad en el desarrollo.

Los **patrones arquitectónicos evolutivos** ofrecen estrategias para transformar sistemas existentes de manera progresiva, minimizando los riesgos y costos asociados con una reescritura completa. En esta sesión, exploraremos algunos de los patrones más utilizados en la modernización de software.

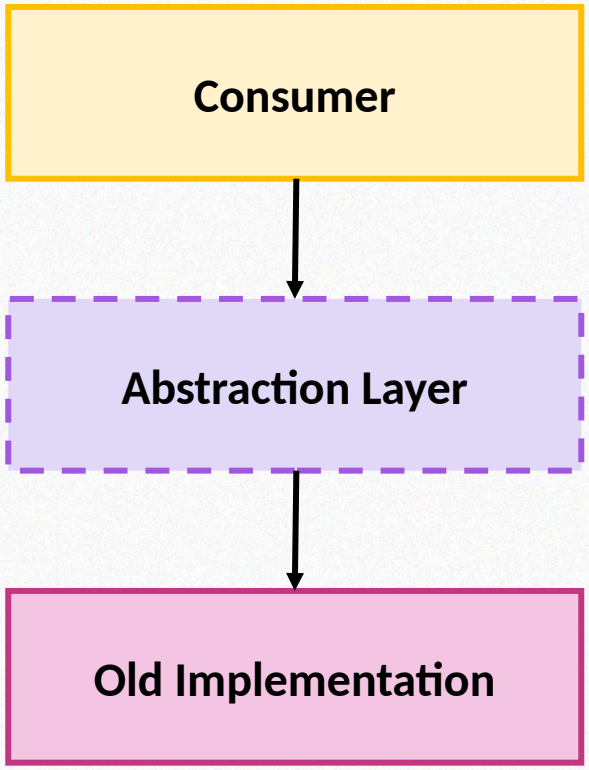


Branch by Abstraction:
Migrar sin interrumpir el desarrollo

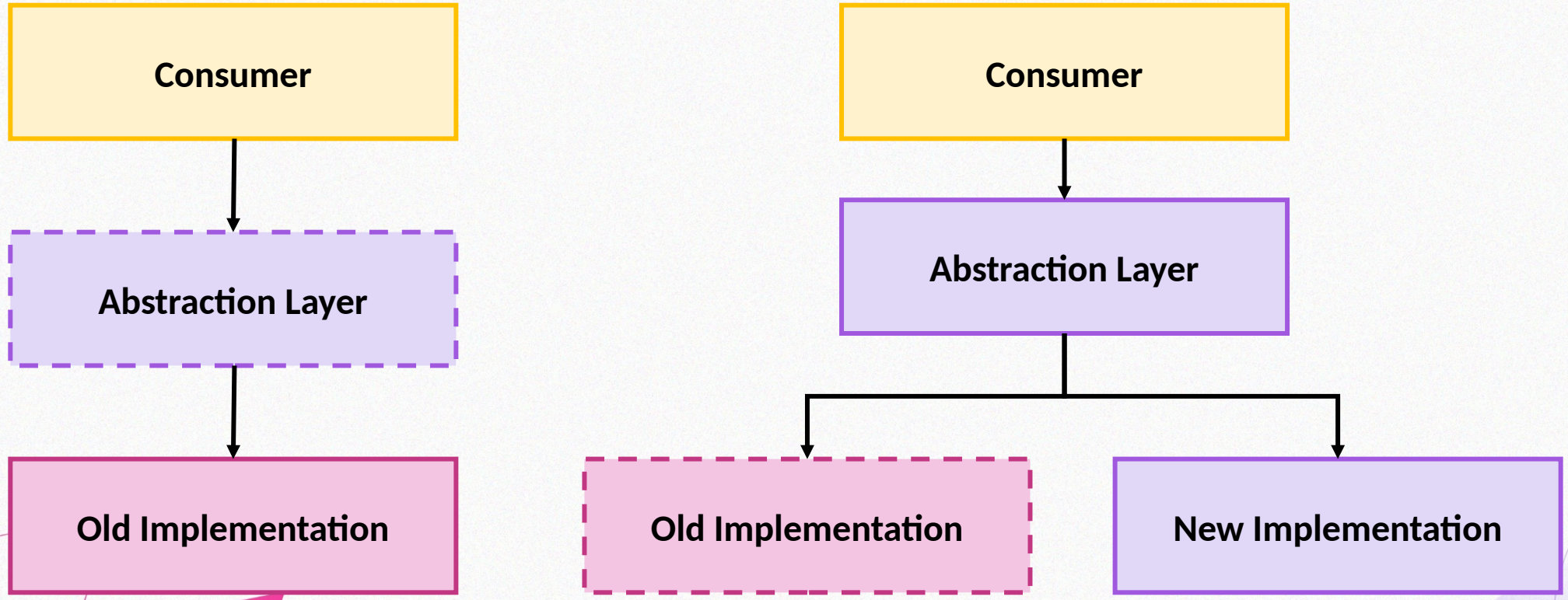
Uno de los mayores desafíos en la refactorización arquitectónica es la coexistencia del código nuevo con el código antiguo. **Branch by Abstraction** es una estrategia que permite hacer cambios profundos en una aplicación sin necesidad de crear una rama separada en el sistema de control de versiones.

¿Cómo funciona?

En lugar de hacer un “fork” de todo el código y trabajar en una versión separada (*lo que podría causar conflictos al integrar los cambios*), **se introduce una abstracción en el código existente** que permite que tanto la implementación antigua como la nueva coexistan temporalmente.



Branch by Abstraction:
Migrar sin interrumpir el desarrollo



Branch by Abstraction:
Migrar sin interrumpir el desarrollo

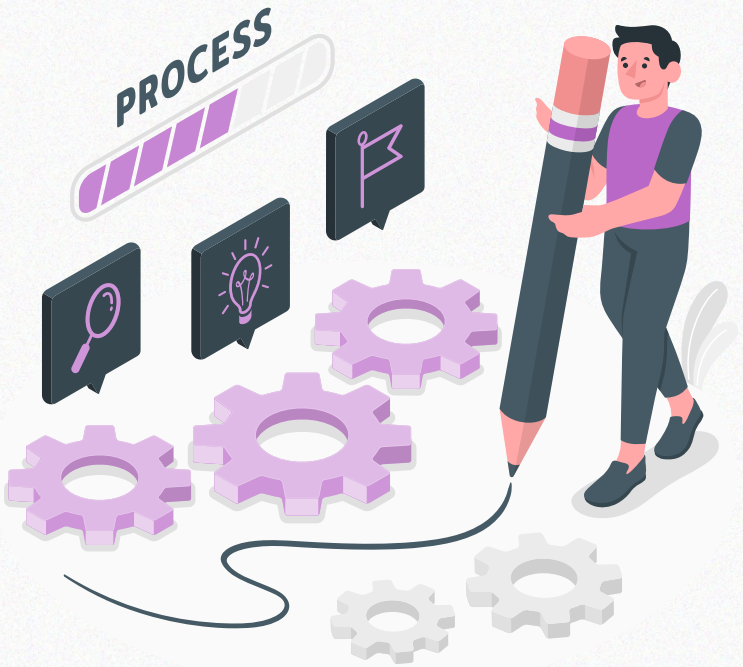
Pasos para aplicar Branch by Abstraction:

1. **Crear una abstracción**: Se introduce una capa de interfaz que encapsula la funcionalidad que queremos reemplazar.

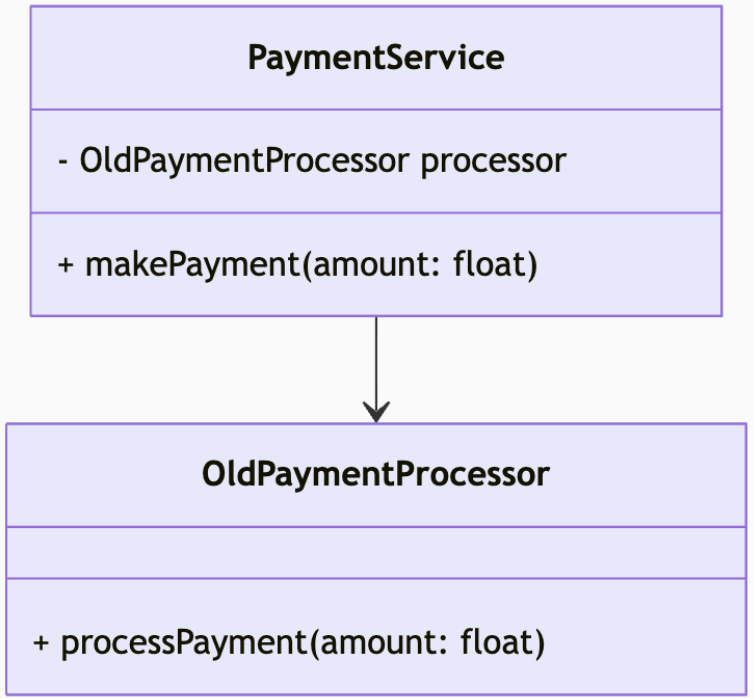
2. **Migrar la implementación sin afectar al sistema**: Se desarrolla una nueva versión de la funcionalidad detrás de la abstracción sin modificar el código que la usa.

3. **Redirigir el código a la nueva implementación**: Una vez que la nueva versión está lista y probada, los puntos de acceso se dirigen a la nueva implementación.

4. **Eliminar la versión antigua**: Una vez validado que todo funciona correctamente, se borra el código antiguo.

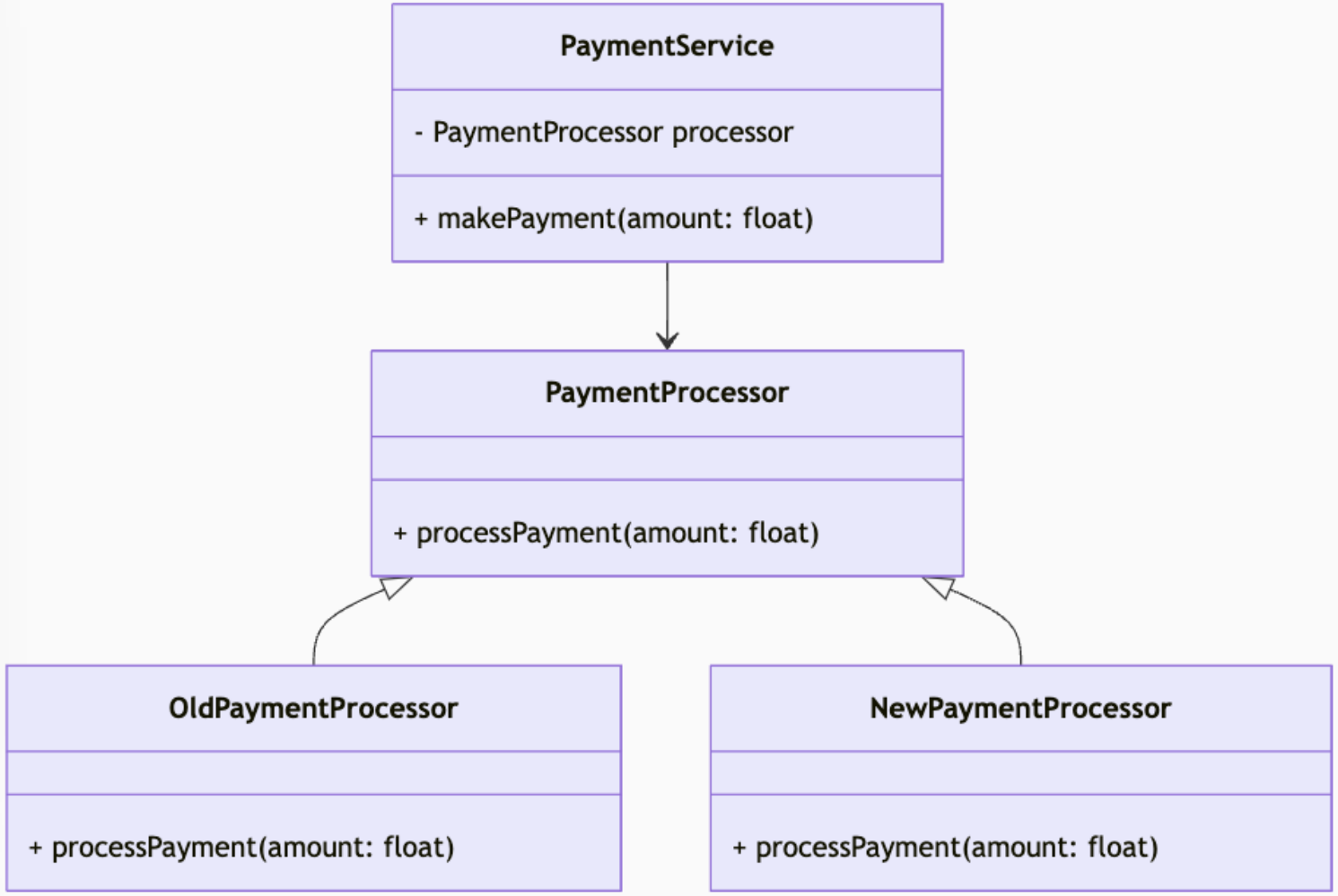


Ejemplo




```python
class OldPaymentProcessor:

    def process_payment(self, amount: float):
        print(f"Procesando con sistema antiguo.")

class PaymentService:

    def __init__(self):
        self.processor = OldPaymentProcessor()

    def make_payment(self, amount: float):
        self.processor.process_payment(amount)

# Uso del sistema

service = PaymentService()
service.make_payment(100.00)
```

Ejemplo


PaymentService
- PaymentProcessor processor
+ makePayment(amount: float)
PaymentProcessor
+ processPayment(amount: float)
OldPaymentProcessor
+ processPayment(amount: float)
NewPaymentProcessor
+ processPayment(amount: float)

Ejemplo

```python
# Interfaz abstracta
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount: float):
        pass

# Implementación antigua
class OldPaymentProcessor(PaymentProcessor):
    def process_payment(self, amount: float):
        print(f"Procesando pago de ${amount} con el sistema antiguo.")

# Nueva implementación
class NewPaymentProcessor(PaymentProcessor):
    def process_payment(self, amount: float):
        print(f"Procesando pago de ${amount} con el NUEVO sistema.")

# Servicio de pago desacoplado
class PaymentService:
    def __init__(self, processor: PaymentProcessor):
        self.processor = processor  # Se usa una abstracción

    def make_payment(self, amount: float):
        self.processor.process_payment(amount)
```

Ejemplo

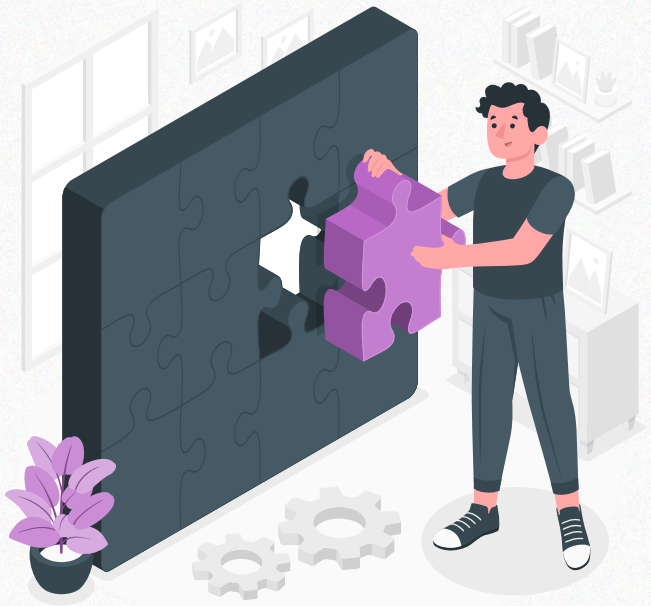
```python
# Uso inicial con la versión antigua
old_processor = OldPaymentProcessor()
service = PaymentService(old_processor)
service.make_payment(100.00)

# Migración a la nueva implementación sin modificar PaymentService
new_processor = NewPaymentProcessor()
service.processor = new_processor  # Migración progresiva
service.make_payment(100.00)
```

Patrón Estrangulador

El **Patrón Estrangulador** (*Strangler Fig Pattern*), introducido por Martin Fowler (2004), es una estrategia para migrar progresivamente de una arquitectura heredada a una nueva sin necesidad de reescribir toda la aplicación desde cero.

¿Cómo funciona?

1. Se identifica un módulo o funcionalidad específica en el sistema monolítico que necesita ser modernizada.
2. Se desarrolla una versión nueva de esa funcionalidad en paralelo con la existente.
3. Se configura un **proxy o API Gateway** para redirigir progresivamente el tráfico a la nueva implementación.
4. Cuando el módulo nuevo está completamente operativo, el módulo viejo se elimina.



Ejemplo

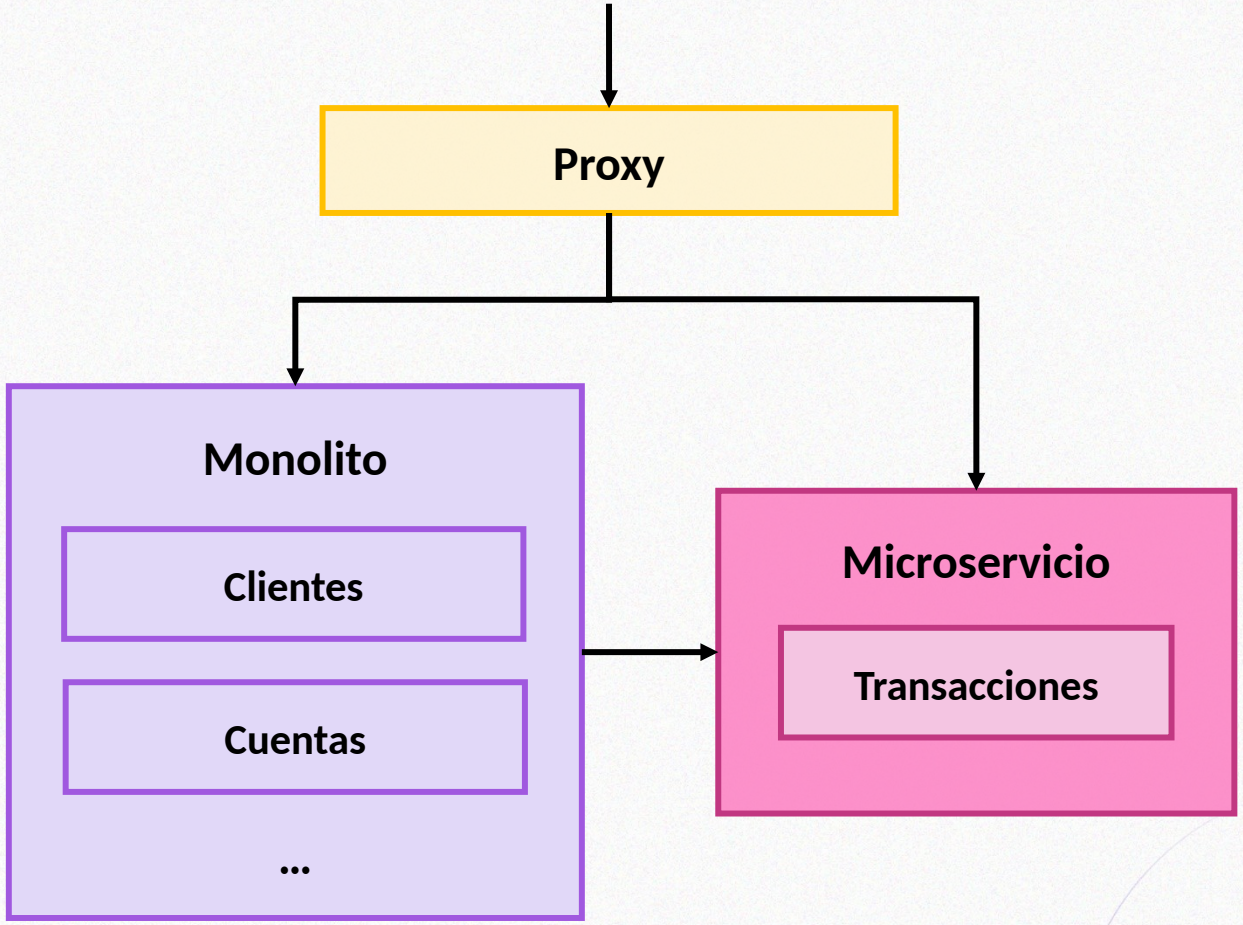
Una empresa de banca en línea tiene un sistema monolítico para gestionar cuentas de clientes.

En lugar de reescribir todo el sistema, comienzan desarrollando un nuevo servicio para la gestión de transacciones y redirigen tráfico hacia él, eliminando gradualmente la versión antigua.

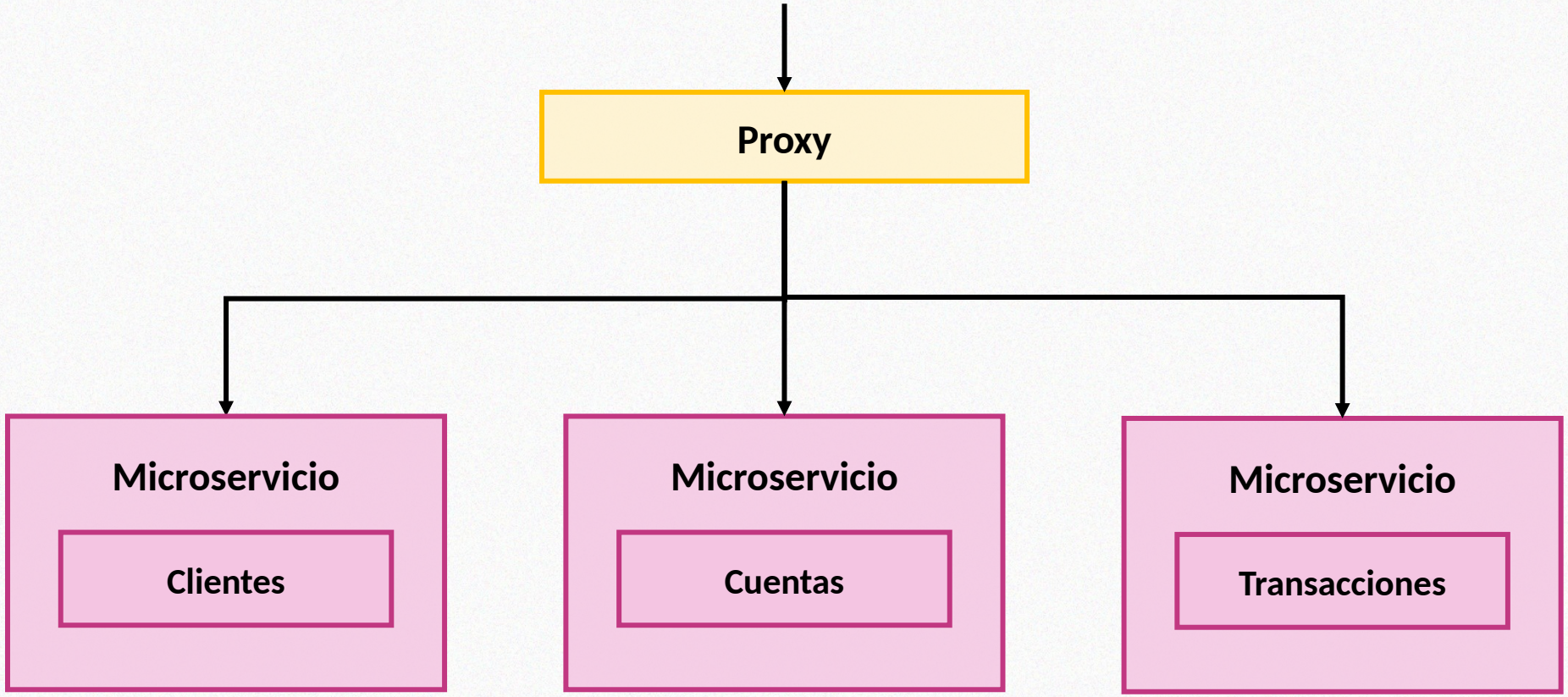
Ejemplo

Una empresa de banca en línea tiene un sistema monolítico para gestionar cuentas de clientes.

En lugar de reescribir todo el sistema, comienzan desarrollando un nuevo servicio para la gestión de transacciones y redirigen tráfico hacia él, eliminando gradualmente la versión antigua.



Ejemplo


Proxy
Microservicio
Clientes
Microservicio
Cuentas
Microservicio
Transacciones

Patrón Estrangulador en Acción

Implementación del Patrón Estrangulador

Para aplicar correctamente este patrón, debemos seguir una serie de pasos estratégicos que aseguren una migración controlada.

Identificar el Módulo a Migrar

Elegir una funcionalidad del sistema monolítico que sea candidata para ser refactorizada.

Para ello se puede usar alguna de las metodologias anteriormente expuestas para una correcta priorización.

Crear un Proxy o API Gateway

Implementar una capa intermedia que redirija tráfico según reglas definidas.

Este proxy permitirá derivar ciertas peticiones al nuevo servicio mientras mantiene las antiguas funcionando.

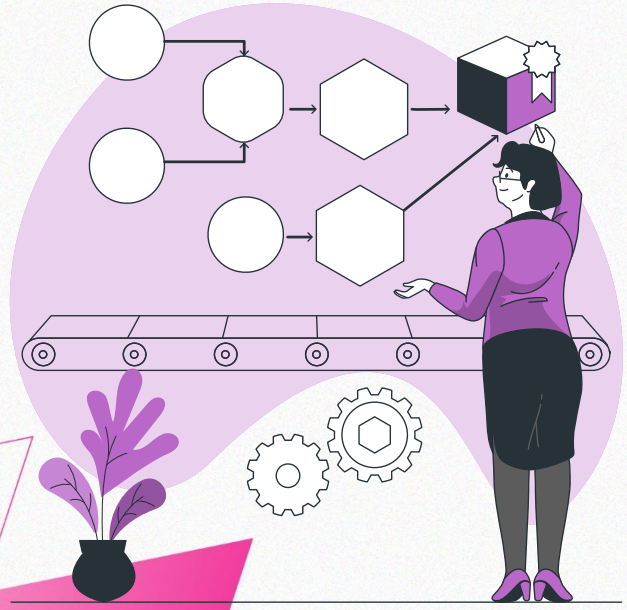
Desarrollar la Nueva Implementación

Crear el nuevo módulo asegurando que sea **modular, independiente y escalable**.

Agregar pruebas automáticas para garantizar compatibilidad con el sistema existente.

Implementación del Patrón Estrangulador

Para aplicar correctamente este patrón, debemos seguir una serie de pasos estratégicos que aseguren una migración controlada.



Redirigir Progresivamente el Tráfico

Usar la API Gateway para que algunas solicitudes sean servidas por la nueva arquitectura.

Realizar pruebas con datos reales y corregir errores antes de continuar la migración completa.

Eliminar el Código Legado Estrangulado

Una vez que la nueva implementación está validada, eliminar el código antiguo.

Mantener monitoreo activo para prevenir problemas en producción.

Ejemplo

Supongamos que una empresa de comercio electrónico tiene un sistema monolítico donde la **gestión de usuarios** está entrelazada con otros módulos como pagos, pedidos y catálogo de productos.

Problema:

• Cualquier cambio en la gestión de usuarios requiere modificar todo el monolito.

• Es difícil escalar el módulo de autenticación sin afectar otros servicios.

Ejemplo

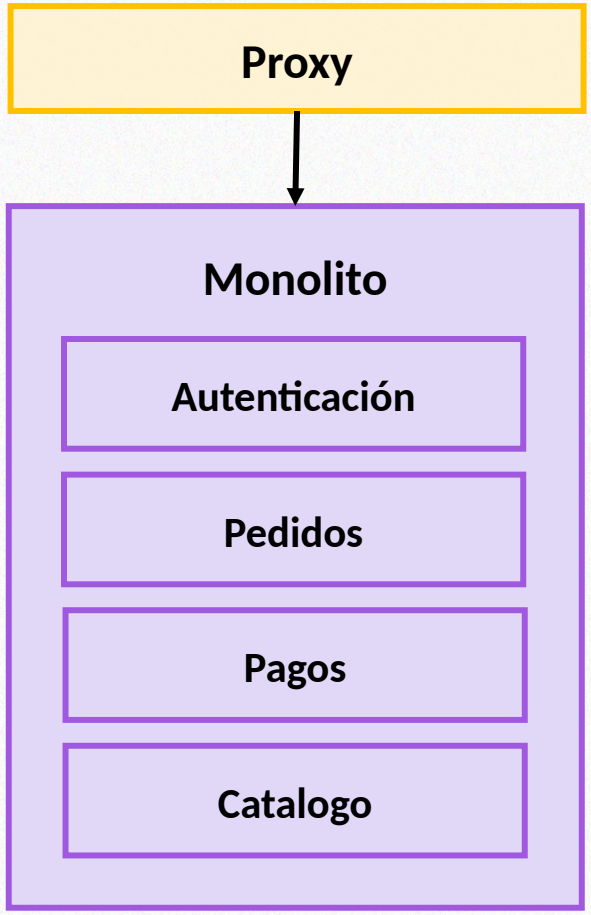
```
class UserAuth:
    def login(self, username, password):
        user = db.find_user(username)
        if not user or not check_password(password,
user.password):
            return {"error": "Invalid credentials"}, 401
        token = generate_session_token(user)
        return {"token": token}

def register(self, username, password):
    if db.user_exists(username):
        return {"error": "User already exists"}, 400
    user = db.create_user(username, hash_password(password))

    return {"message": "User created successfully"}
```

Ejemplo

```
class UserAuth:
    def login(self, username, password):
        user = db.find_user(username)
        if not user or not check_password(password, user.password):
            return {"error": "Invalid credentials"}, 401
        token = generate_session_token(user)
        return {"token": token}

def register(self, username, password):
    if db.user_exists(username):
        return {"error": "User already exists"}, 400
    user = db.create_user(username, hash_password(password))

    return {"message": "User created successfully"}
```

Ejemplo

```
routes:
  - name: authentication
    paths: /auth
    upstream:
      url: http://monolith/auth
```

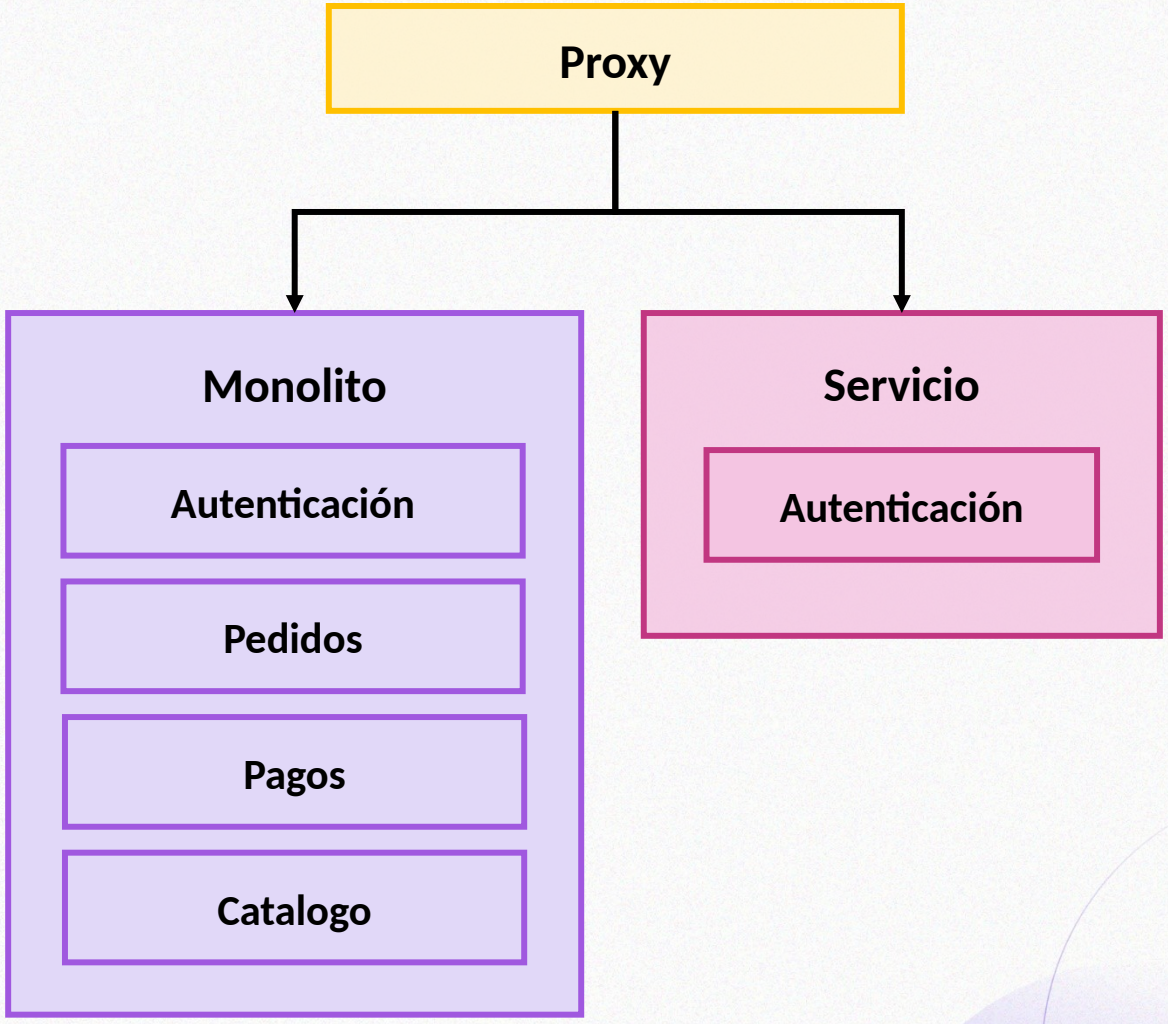


Ejemplo

```python
from flask import Flask, request, jsonify
import jwt, datetime

app = Flask(__name__)
SECRET_KEY = "mysecret"

users_db = {"user1": "hashed_password"}  # Simulación de DB

@app.route("/login", methods=["POST"])
def login():
    data = request.json
    username, password = data["username"], data["password"]

    if username not in users_db or users_db[username] != password:
        return jsonify({"error": "Invalid credentials"}), 401

    token = jwt.encode({"user": username, "exp": datetime.datetime.utcnow() + 
    return jsonify({"token": token})

if __name__ == "__main__":
    app.run(port=5001)
```

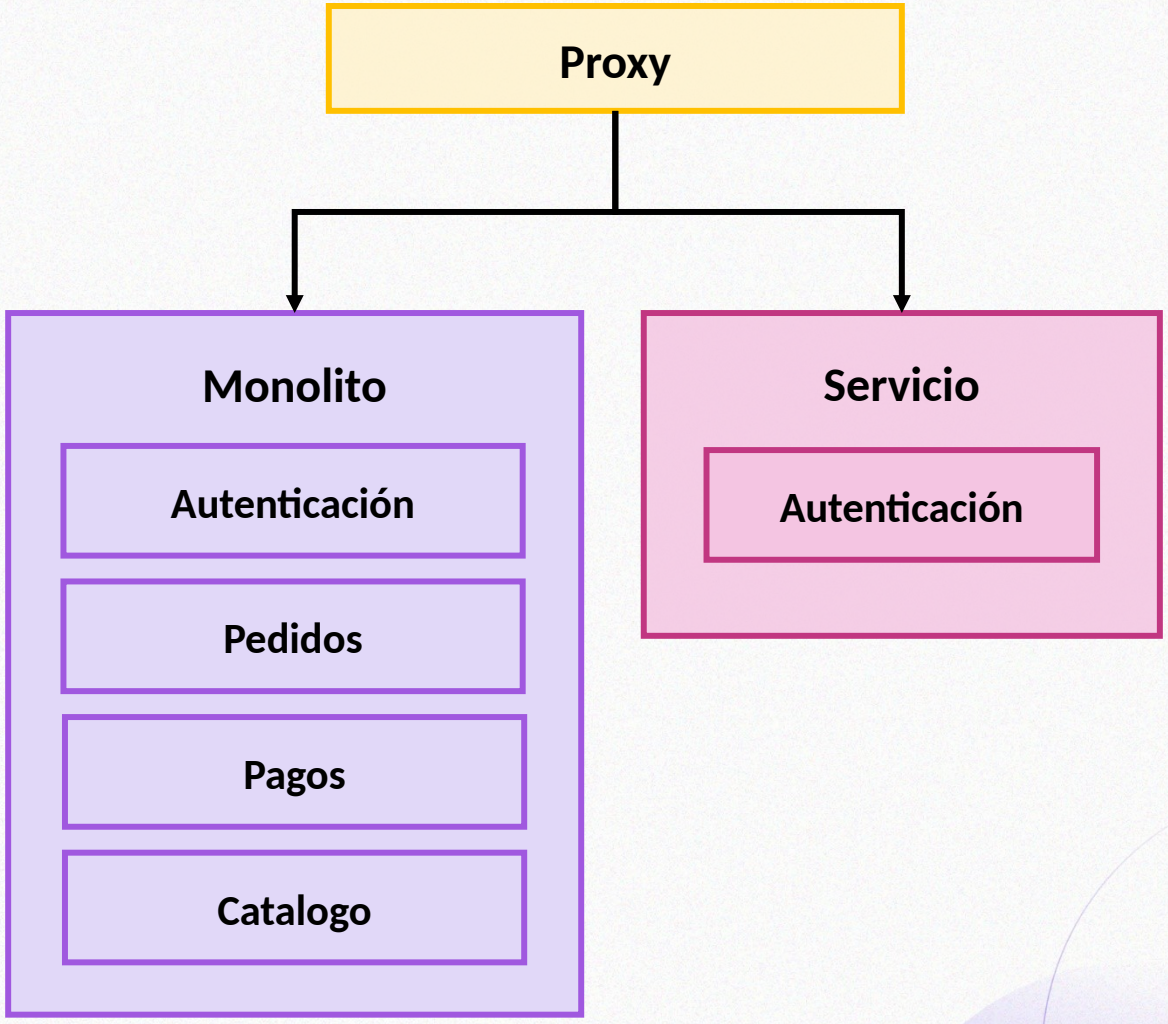
Ejemplo

```yaml
# api-gateway.yaml

routes:
  - name: authentication
    paths: /auth
    upstreams:
      - url: http://monolith/auth
        weight: 80
      - url: http://auth_service:5001
        weight: 20
```



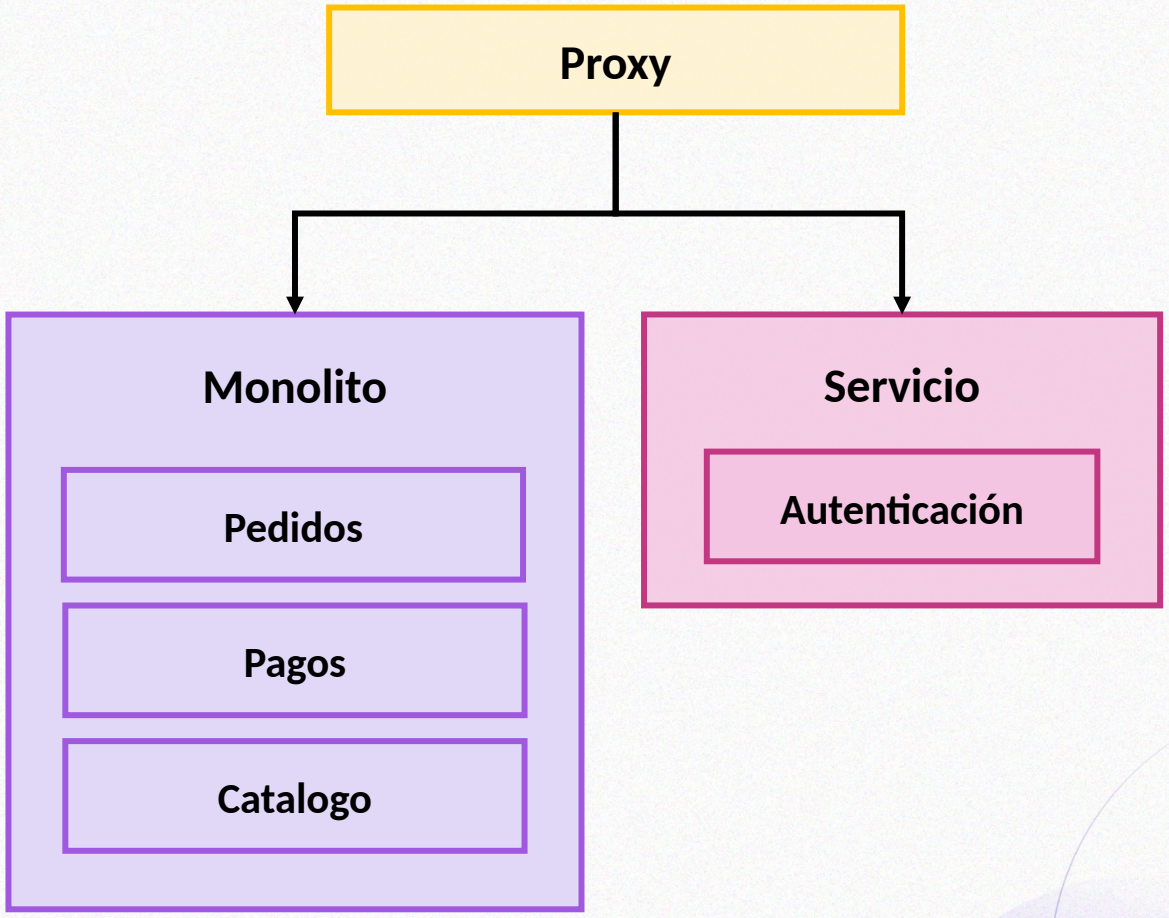
Ejemplo

```yaml
# api-gateway.yaml

routes:
  - name: authentication
    paths: /auth
    upstreams:
      - url: http://monolith/auth
        weight: 20
      - url: http://auth_service:5001
        weight: 80
```



Ejemplo

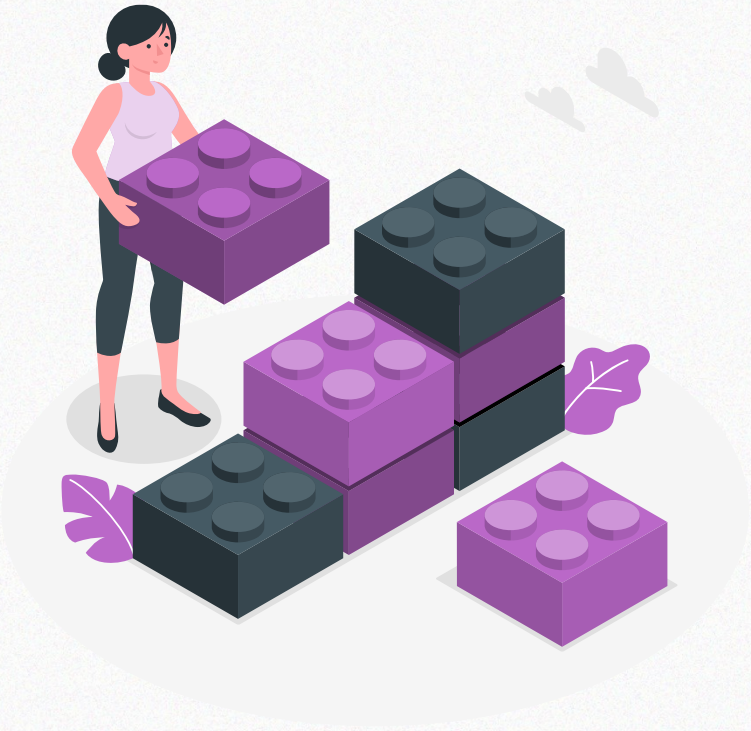
```yaml
# api-gateway.yaml

routes:
  - name: authentication
    paths: /auth
    upstreams:
      - url: http://auth_service:5001
```



Estrategia de Modularización y Separación de Concerns

La modularización es el proceso de dividir una aplicación en componentes independientes que pueden evolucionar por separado sin afectar la funcionalidad global.

En software, esta estrategia se conoce como **Separación de Responsabilidades (Separation of Concerns, SoC)**.



Problema Clásico: El Monolito con Dependencias Cruzadas

Ejemplo

Imagina que tenemos una aplicación monolítica para una tienda en línea con los siguientes módulos:

- **Usuarios**
- **Pedidos**
- **Pagos**
- **Inventario**

```python
class OrderService:
    def create_order(self, user_id, product_id, payment_info):
        user = UserRepository.get_user(user_id)
        if not user.is_active:
            raise Exception("User not active")

        inventory = InventoryService()
        if not inventory.is_available(product_id):
            raise Exception("Product not available")

        payment = PaymentService()
        if not payment.process_payment(payment_info):
            raise Exception("Payment failed")

        return "Order Created"
```

Problemas detectados:

- **Alta dependencia entre servicios:** Si se cambia un método en InventoryService, es probable que OrderService falle.
- **Baja reutilización de código:** OrderService no puede usarse sin InventoryService ni PaymentService.
- **Dificultad para escalar:** Cada cambio en OrderService podría requerir modificar varios módulos.

En una arquitectura mal diseñada, estas capas están **fuertemente acopladas**:

Ejemplo

Imagina que tenemos una aplicación monolítica para una tienda en línea con los siguientes módulos:

- **Usuarios**
- **Pedidos**
- **Pagos**
- **Inventario**

```python
class OrderService:
    def create_order(self, user_id, product_id, payment_info):
        user = UserRepository.get_user(user_id)
        if not user.is_active:
            raise Exception("User not active")

        inventory = InventoryService()
        if not inventory.is_available(product_id):
            raise Exception("Product not available")

        payment = PaymentService()
        if not payment.process_payment(payment_info):
            raise Exception("Payment failed")

        return "Order Created"
```

Problemas detectados:

- **Alta dependencia entre servicios:** Si se cambia un método en InventoryService, es probable que OrderService falle.
- **Baja reutilización de código:** OrderService no puede usarse sin InventoryService ni PaymentService.
- **Dificultad para escalar:** Cada cambio en OrderService podría requerir modificar varios módulos.

En una arquitectura mal diseñada, estas capas están **fuertemente acopladas**:

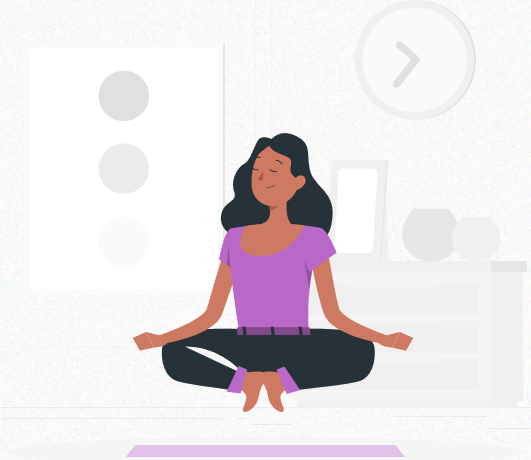
Aplicando Refactorización en Proyectos Reales

El Viaje de YourFinFreedom a Microservicios

En una mañana como cualquier otra, Sofía, CTO de **YourFinFreedom**, una fintech en rápido crecimiento, se reunió con su equipo de ingeniería. Sobre la mesa estaban los reportes de los últimos meses:

El sistema era un **monolito gigante**, construido en los primeros días de la startup. Había funcionado bien cuando solo tenían unos miles de usuarios, pero ahora, con **millones de clientes** usando la plataforma, cada cambio se volvía un dolor de cabeza. Peor aún, los **equipos estaban bloqueados**, ya que cualquier modificación en una parte del código podía romper otra inesperadamente.

- **Tiempo promedio de despliegue:** 4 días.
- **Tasa de fallos en producción:** 15% de los cambios generaban errores inesperados.
- **Escalabilidad:** Cada pico de usuarios causaba caídas en el sistema.



¿Qué modelo arquitectónico podría ayudar a YourFinFreedom a resolver estos problemas?

Evaluación con ATAM: Entendiendo el Problema Antes de Actuar

El equipo organizó una sesión de ATAM para evaluar los trade-offs de su arquitectura actual y posibles soluciones.

Se definieron los siguientes **atributos de calidad** clave:

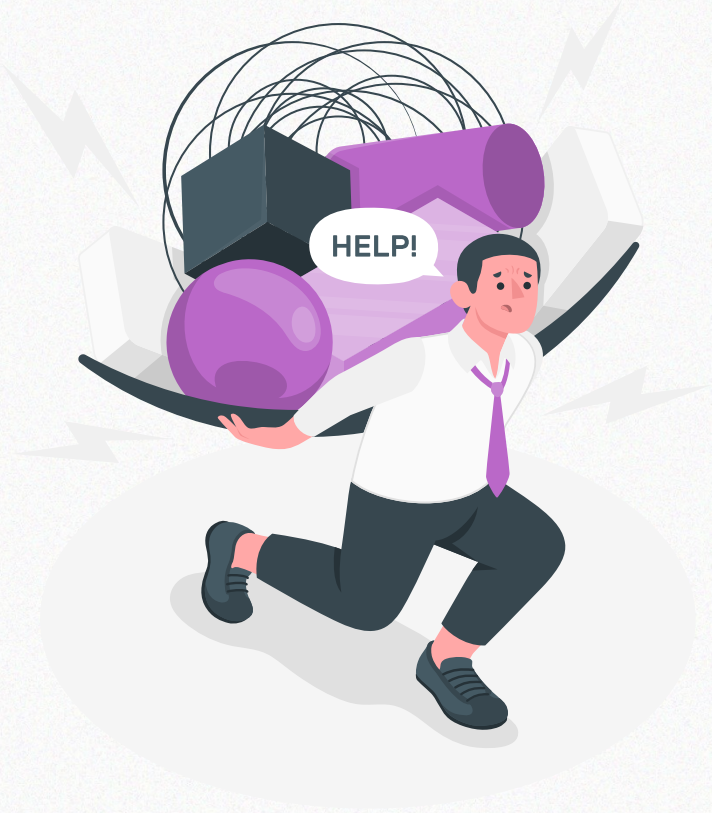
Atributo de calidad	Importancia (Alto, Medio, Bajo)	Problema actual
Escalabilidad	🔥 Alto	Cada pico de usuarios genera caídas.
Mantenibilidad	🔥 Alto	Cada cambio impacta múltiples módulos, causando fallos.
Despliegue rápido y seguro	🔥 Alto	Desplegar una nueva funcionalidad toma días.
Rendimiento	⚡ Medio	Algunas consultas son lentas, pero no es el problema más grave.
Tolerancia a fallos	⚡ Medio	No es crítica ahora, pero debería mejorar.

¿Qué queremos mejorar con la migración?

La Migración - El Camino al Caos

Con entusiasmo, el equipo inició la migración. **Dividieron el monolito en múltiples servicios**, asignaron equipos a cada uno y establecieron una estrategia de despliegue.

Pronto, pasó de **1 gran aplicación monolítica a 20 microservicios independientes**.



**Todo parecía ir bien…
hasta que aparecieron
los problemas.**

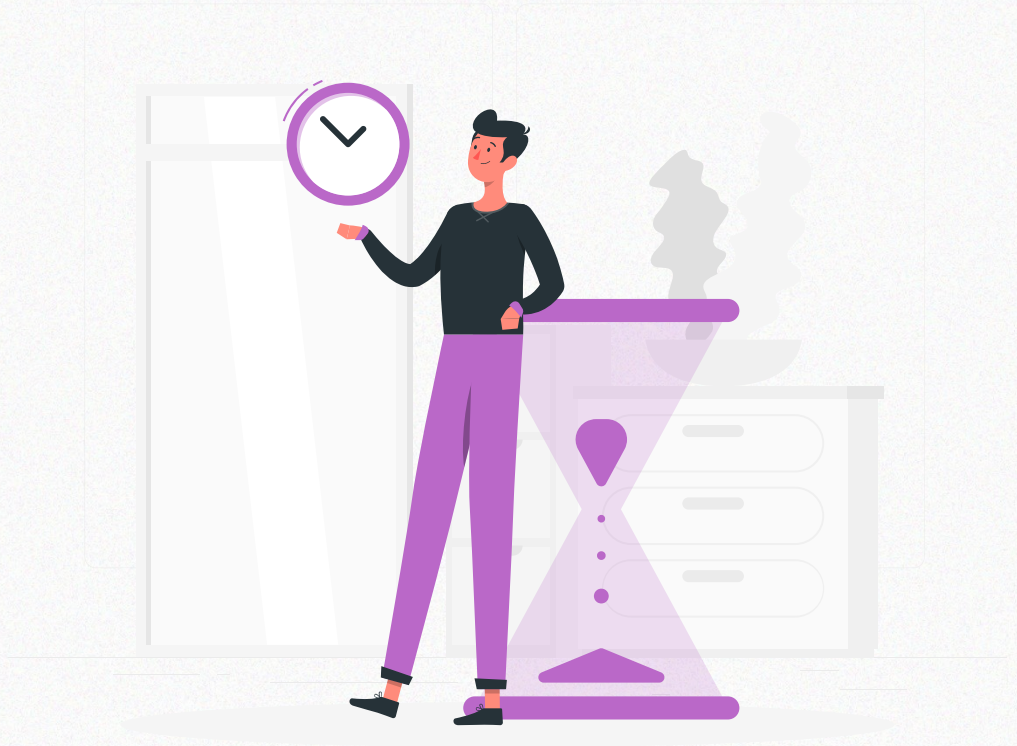
Primer síntoma:
Latencias inesperadas

El sistema ahora era **distribuido**, pero algo extraño pasaba. **Las consultas que antes tomaban 200ms ahora tardaban 2 segundos**.

¿Qué estaba pasando?

- **En un monolito**, los módulos se comunican en memoria, lo que es muy rápido.

- **En microservicios**, la comunicación ahora dependía de llamadas de red (REST), lo que añadía latencia.



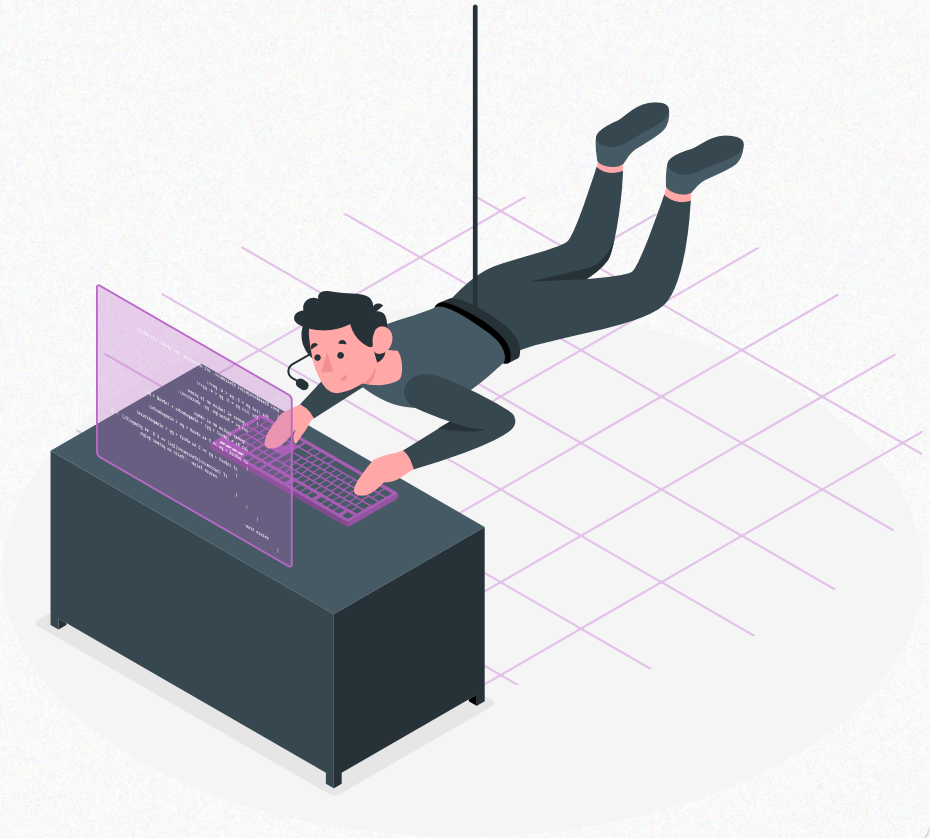
Segundo síntoma: Dependencias ocultas y el “Big Ball of Mud”

A medida que más microservicios se ponían en producción, el equipo notó un **nuevo problema**:
Un pequeño cambio en un servicio a veces rompía otros servicios inesperadamente.

¿Qué estaba pasando?

- En un monolito, el acoplamiento es evidente porque el código está en un solo lugar.

- En microservicios, el acoplamiento puede **ocultarse** en las dependencias de red.



Tercer síntoma:
La complejidad operativa explotó

Antes, desplegar el monolito era **complicado**, pero al menos estaba **todo en un solo servidor**. Ahora, con **20 microservicios**, las fallas eran impredecibles.

¿Qué estaba pasando?

- **El costo de la complejidad operativa en arquitecturas distribuidas.**

- Se debe invertir en **observabilidad**, trazabilidad y automatización.

La Optimización
Encontrando el Equilibrio

Así que analizaron métricas clave:

1. **Tiempo promedio de respuesta por servicio.**
2. **Número de dependencias entre servicios.**
3. **Cantidad de fallos en despliegues.**

La Optimización
Encontrando el Equilibrio

Se decidio:

- Fusionar servicios que estaban demasiado fragmentados.
- Reorganizar los servicios en **dominios bien definidos**.
- Aplicar el principio **“Smart endpoints, dumb pipes”**, para que la lógica viva en los servicios y no en la comunicación entre ellos.

Primer síntoma: Latencias inesperadas

- Se introdujo **mensajería asíncrona con Kafka**, para evitar que cada servicio tuviera que esperar la respuesta de otro antes de continuar.

Segundo síntoma: Dependencias ocultas y el “Big Ball of Mud”

- Se introdujo el patrón **Database-per-Service**, asegurando que cada servicio tuviera su propia base de datos.
- Se implementó **APIs bien definidas**, en lugar de permitir accesos directos a datos entre servicios.

Tercer síntoma: La complejidad operativa explotó

- Se integró **OpenTelemetry** para capturar métricas y rastrear solicitudes a través de servicios.
- Se crearon **dashboards en Grafana y Prometheus** para visualizar tiempos de respuesta y cuellos de botella.

NETFLIX

Netflix Technology Blog in Netflix TechBlog

Jan 10, 2024 · 9 min read

Rebuilding Netflix Video Processing Pipeline with Microservices

https://netflixtechblog.com